

1. [Xna0095-Preface](#)
2. [Xna0100-Getting Started](#)
3. [Xna0102-What is C# and Why Should You Care](#)
4. [Xna0104-What is OOP and Why Should You Care?](#)
5. [Xna0106-Encapsulation in C#](#)
6. [Xna0108-Inheritance in C#](#)
7. [Xna0110-Polymorphism Based on Overloaded Methods](#)
8. [Xna0112-Type Conversion, Casting, and Assignment Compatibility](#)
9. [Xna0114-Runtime Polymorphism through Class Inheritance](#)
10. [Xna0116-Runtime Polymorphism and the Object Class](#)
11. [Xna0118-The XNA Framework and the Game Class](#)
12. [Xna0120-Moving Your Sprite and using the Debug Class](#)
13. [Xna0122-Frame Animation using a Sprite Sheet](#)
14. [Xna0124-Using Background Images and Color Key Transparency](#)
15. [Xna0126-Using OOP - A Simple Sprite Class](#)
16. [Xna0128-Improving the Sprite Class](#)
17. [Xna0130-Collision Detection](#)
18. [Xna0132-A Simple Game Program with On-Screen Text](#)

Xna0095-Preface

This module is the first in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin Community College in Austin, TX.

Revised: Wed May 04 09:58:08 CDT 2016

This page is part of a Book titled [XNA Game Studio](#).

Preface to XNA Game Studio Collection

This module is the first in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

The modules were originally published for use with *XNA 3.1* , were upgraded for use with *XNA 4.0* , and as of May 2016 are being upgraded for use with *XNA 4.0 Refresh* .

-end-

Xna0100-Getting Started

Learn how to download, install, and test Microsoft's XNA Game Studio.

Revised: Fri May 19 17:26:11 CDT 2017

This page is part of a Book titled [XNA Game Studio](#).

Table of Contents

- [Table of Contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
- [Preview](#)
- [Download and installation instructions](#)
 - [Introduction](#)
 - [Software upgrade](#)
 - [XNA installation instructions](#)
 - [Cleaning off the old software](#)
 - [Downloading Microsoft Visual C# 2010 Express Edition](#)
 - [Preparing the ISO file for software installation](#)
 - [Installing Microsoft Visual C# 2010 Express Edition](#)
 - [Downloading and installing XNA 4.0 Refresh](#)
- [Test your installation](#)
 - [Skeleton code](#)
 - [Run your program](#)
 - [Run your program outside the IDE](#)
- [Run my program](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

The modules were originally published for use with *XNA 3.1* , were upgraded for use with *XNA 4.0* , and as of May 2016 are being upgraded for use with *XNA 4.0 Refresh* .

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures while you are reading about them.

Figures

- [Figure 1](#). XNA 4.0 IDE.
- [Figure 2](#). XNA game window.
- [Figure 3](#). Project file structure.

Preview

According to Microsoft, *"Microsoft XNA Game Studio 4.0 makes it easier than ever to create great video games for Windows-based PCs, Xbox 360 consoles, and Windows Phone."*

The purpose of this module is to help you get started programming using the XNA Game Studio.

Download and installation instructions

The download and installation information regarding the Microsoft website changes frequently. The information in this section is current as of May 2016 (*as updated in May 2017*) .

Introduction

I typically teach this course once each year in the Summer session via Distance Learning. For the past several years, students have reported that it has become increasingly difficult to find and download the necessary Microsoft software for the course. In preparation for the Summer 2016 session, I decided to do the following:

- Clean the old software off of my computer.
- Download and install new software.
- Document the process for the benefit of students who may need to do the same.

Software upgrade

I also decided to upgrade from XNA 4.0 to a newer product named [Microsoft XNA Game Studio 4.0 Refresh](#) . Here is what Microsoft has to say about the newer product:

Note: *Microsoft XNA Game Studio 4.0 Refresh updates XNA Game Studio 4.0 to fix bugs and add support for developing games that target Windows Phone OS 7.1 and developing games in Visual Basic.*

This course doesn't address Visual Basic or Windows Phone, but bug fixes are always welcome.

XNA installation instructions

Microsoft's [installation instructions](#) for the new software read as follows:

1. *Install Microsoft Visual Studio 2010.*
2. *Obtain the latest updates for Visual Studio from Microsoft Update.*
3. *Download and run the Microsoft XNA Game Studio 4.0 Refresh installer.*
4. *Follow the setup instructions.*

As you can see from the installation instructions, *Microsoft Visual Studio 2010* is required. This has been a big part of the student's problems in downloading and installing the software in previous semesters.

However, a [different page](#) indicates that *Microsoft Visual C# 2010 Express Edition* will suffice in place of *Microsoft Visual Studio 2010* . That page reads as follows:

XNA Game Studio 4.0 Refresh works with any of the following Microsoft Visual Studio 2010 products.

- *Microsoft Visual Studio 2010 Express for Windows Phone*
- *Microsoft Visual C# 2010 Express Edition*
- *Microsoft Visual Studio 2010 Professional Edition*

I will have more to say about this later.

Cleaning off the old software

I began by using the software removal procedures in the Control Panel to remove three different *Microsoft XNA* items and one item titled *Microsoft Visual C# 2010 Express - ENU*. At that point, my computer had no XNA capability and no Visual C# capability.

Downloading Microsoft Visual C# 2010 Express Edition

A big part of the student's problems in previous semesters has been the difficulty of finding and downloading *Microsoft Visual C# 2010 Express Edition*. The only way that I have found to download it is by clicking the following link: [Visual Studio 2010 Express All-in-One ISO](#) (see update below) and saving the file named **VS2010Express1.iso** that is downloaded.

Note:

Author's update: May 19, 2017

In preparation for the Summer 2017 teaching session, I needed to install Visual C# Express 2010 on my Dell XPS 14 Z laptop computer running Windows 7 Home Premium in order to use **XNA 4.0 Refresh Game Studio** for the course. This required downloading the file named **VS2010Express1.iso** mentioned above followed by some additional steps that are explained later in this module.

While I still had a copy of the file named **VS2010Express1.iso** from the previous year that I could have used, I decided to start from scratch, download, and install the software on my Dell XPS laptop to confirm the installation process described in this module. In so doing, I discovered that the following link given above was broken:

[Visual Studio 2010 Express All-in-One ISO](#)

Apparently Microsoft is discouraging the use of Visual C# 2010 and is recommending [Visual Studio Community](#) as a replacement. However, rather than to face the requirement of testing a new software suite, I decided to stick with what I knew would work for all of the instructional material and assignments for this course and began searching for an alternative source for the software. A Google search identified the following MediaFire site as an alternative download site for this iso file:

<http://www.mediafire.com/file/bbouvxxhumaho8qr/VS2010Express1.iso>

While I cannot guarantee the integrity of the **MediaFire** site, **Norton SafeSearch** showed the site to be safe. I downloaded and saved the file. Then I performed a Norton virus scan on the file, which showed the file to contain "No Threats." The size of the file downloaded from **MediaFire** was exactly the same as the file that I had downloaded from Microsoft a year earlier (710,304 KB).

Having downloaded the iso file, I successfully installed and tested **Visual C# Express 2010** and **XNA 4.0 Refresh** on my Dell XPS 14 Z laptop running Windows 7 Home Premium according to the instructions provided in the remainder of this module. Thus, the download and installation process is confirmed for the Summer session of 2017.

The file named **VS2010Express1.iso** not only contains *Microsoft Visual C# 2010 Express Edition* , it also contains the express editions for several other languages as well. However, it is **not** an installable product at this point.

Preparing the ISO file for software installation

There is more than one way to install the software in the ISO file. You might benefit by doing some online research into how to install the software in an ISO file. Here is how I did it with Windows 7:

1. Insert a blank DVD in the DVD burner. (*A blank CD would probably also work but I didn't have one.*)
2. Right-click on the ISO file.
3. Select **Open With...**
4. Select **Windows Disk Image Burner**
5. Follow the disk burning instructions.
6. Remove the DVD from the burner when complete and label it *Visual Studio 2010 Express* or something similar.

The resulting DVD contains an installable version of *Visual C# 2010 Express* and some other programs as well.

Installing Microsoft Visual C# 2010 Express Edition

Insert the DVD in the DVD reader and select **Setup.hta** . You should then be given a choice as to which of the express edition programs you want to install.

Select *Visual C#* and follow the installation instructions.

When the installation is complete, you should be able to see *Microsoft Visual C# 2010 Express* if you open the **Start** menu and enter 2010 in the search box.

At this point, you can run the program, open the **Help** menu, and select *Check for Updates* to satisfy the second item under [XNA installation instructions](#).

Downloading and installing XNA 4.0 Refresh

This is the easy part. Click [here](#) to visit a page where you can download the setup file named **XNAGS40_setup.exe** for *Microsoft XNA Game Studio 4.0 Refresh* . Download, save, double click the file, and follow instructions to install the software.

When the installation is complete, XNA should be merged with Visual C# and you should be able to begin creating the programs for the course.

I have confirmed that all of the assignment and sample program for this course, which were originally written using *XNA 4.0* execute properly under *XNA 4.0 Refresh* .

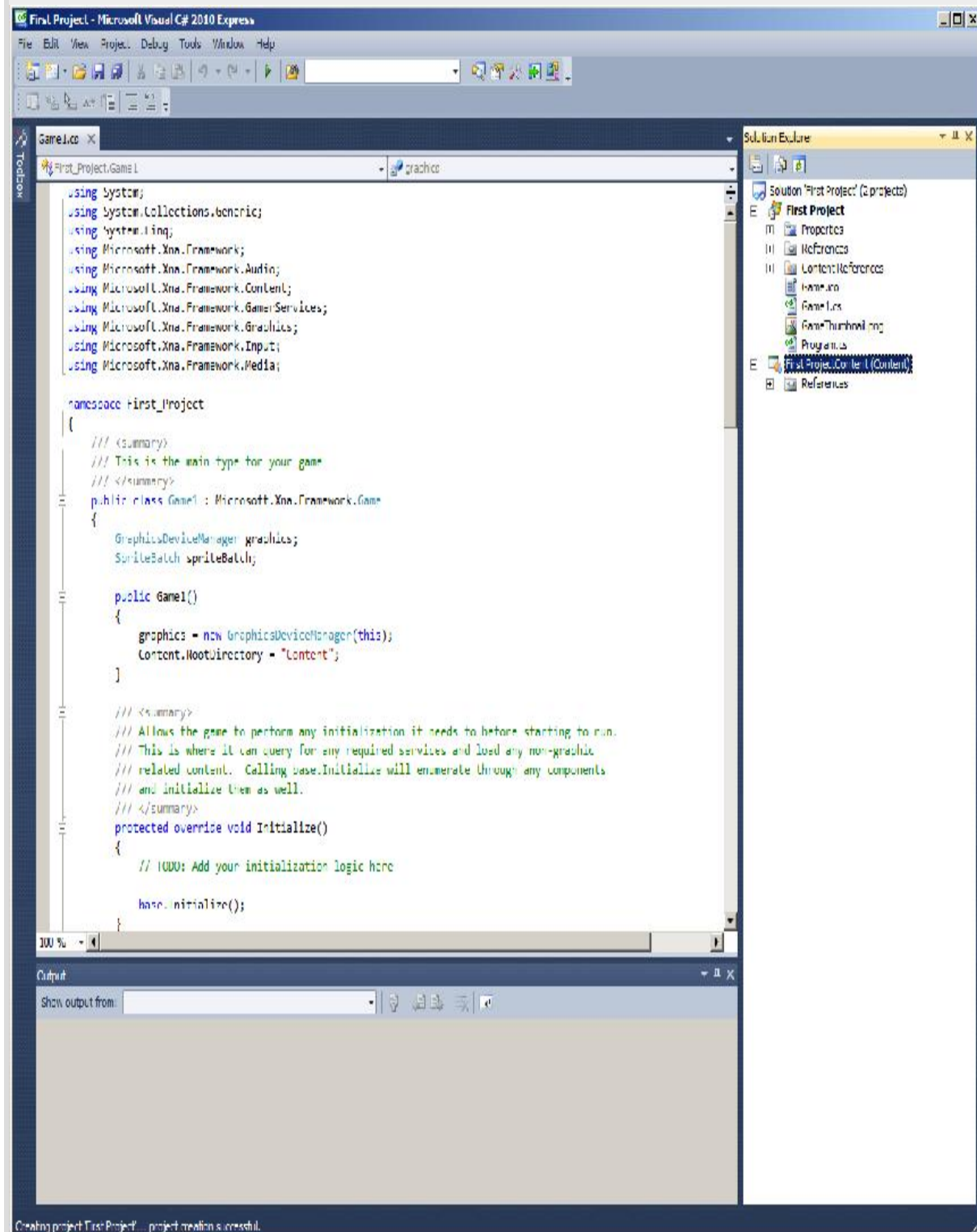
Test your installation

Once you have the software installed, start Visual C# from the Windows **Start** menu. Pull down the **File** menu and select **New Project** . A dialog box will appear. Select "*XNA Game Studio 4.0*" on the left side of the dialog and select "*Windows Game (4.0)*" on the right side of the dialog."

Name your project **First Project** , specify a location for your new project and click the **OK** button. An IDE should appear looking something like [Figure 1](#) . (Note that [Figure 1](#) has been reduced, which causes the text to be difficult to read.)

Note:

Figure 1 . XNA 4.0 IDE.



Skeleton code

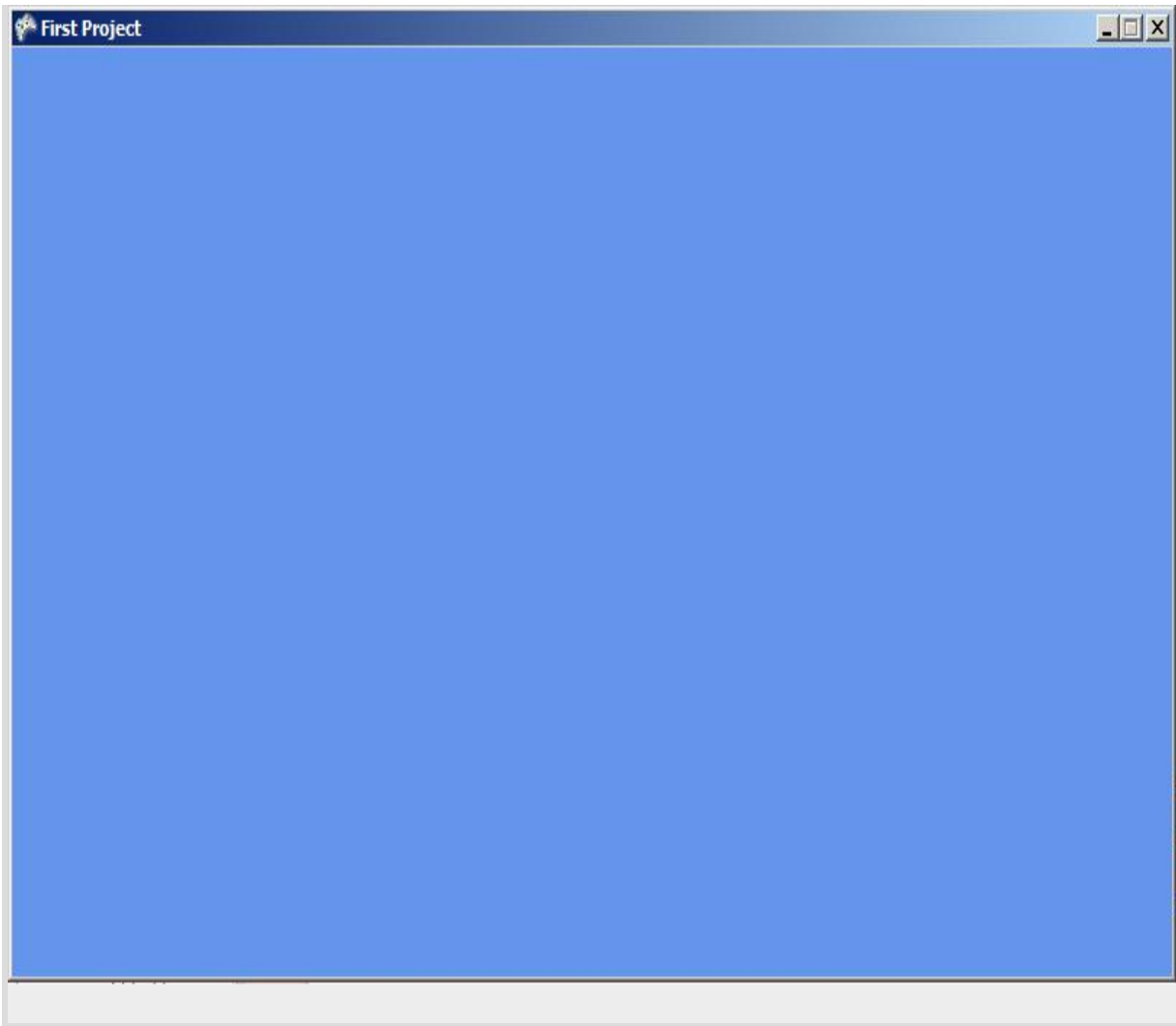
Skeleton code for a Windows game project will have been created in the editor portion of the IDE. I will explain this code in detail in a future module.

Run your program

Run your program by selecting **Start Debugging** on the **Debug** menu. If everything is working properly, the game window shown in [Figure 2](#) should appear on your computer screen.

Note:

Figure 2 . XNA game window.



Run your program outside the IDE

Once you have run your program successfully from inside the IDE, your project file structure should look similar to [Figure 3](#).

Note:

Figure 3 . Project file structure.



At that point, you should be able to double click the exe file highlighted in [Figure 3](#) and cause your program to run.

Congratulations. You're now ready to start learning how to program using XNA.

Run my program

Click [here](#) to download a zip file containing my version of the program. Extract the folder named **First Project** from the zip file and save it somewhere on your disk. Start **Visual C# 2010 Express** and select **Open Project...** from the **File** menu. Navigate to the project folder and select the file named **First Project.sln** . This should cause the project to open and be ready to run or debug as described above.

Many of the modules in this collection will provide my version of a project that can be downloaded and run in this manner.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Xna0100-Getting Started

- File: Xna0100.htm
- Published: 02/24/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Xna0102-What is C# and Why Should You Care
Learn why you need to care about C#.

Revised: Wed May 04 14:28:53 CDT 2016

This page is part of a Book titled [XNA Game Studio](#).

Table of Contents

- [Table of Contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Listings](#)
- [Technical level of the course](#)
 - [I will assume...](#)
 - [I will also assume...](#)
- [What is C#?](#)
- [Why should you care about C#?](#)
 - [Complex IDE](#)
 - [Is there a simpler way?](#)
 - [Fully object oriented](#)
 - [Massive and complex documentation package](#)
 - [Indirection](#)
 - [C# is the language of XNA](#)
- [Run the program](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin

Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

An earlier module titled [Getting Started](#) provided information on how to get started programming with Microsoft's XNA Game Studio.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following link to easily find and view the listing while you are reading about it.

Listings

- [Listing 1](#). Hello World in C#.

Technical level of the course

This is not a beginning programming course. The official prerequisite for this course is a course in programming fundamentals using either C++ or Python. Those courses use C++ or Python strictly as procedural programming languages. The courses are currently taught using very simple IDEs.

I will assume...

I will assume that students in this course already know how to program using either C++ or Python as a procedural programming language and that they already understand such fundamental concepts as programming logic, functions, parameter passing, etc.

I will also assume...

At the same time I will also assume that the students in this course know very little if anything about the following topics:

- The use of a complex IDE.
- The use of any programming language other than either Python or C++ solely as a procedural programming language.
- Object-oriented programming.
- Pointers or any other form of indirection.
- Large-scale programming documentation.

These are all topics that will be important in learning how to make effective use of the Microsoft XNA Game Studio software.

What is C#?

According to Wikipedia,

Note:

"C# (pronounced "see sharp") is a multi-paradigm programming language encompassing imperative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines. It was developed by Microsoft within the .NET initiative and later approved as a standard by Ecma (ECMA-334) and ISO (ISO/IEC 23270). C# is one of the programming languages designed for the Common Language Infrastructure."

"C# is intended to be a simple, modern, general-purpose, object-oriented programming language.[3] Its development team is led by Anders Hejlsberg, the designer of Borland's Turbo Pascal, who has said that its object-oriented syntax is based on C++ and other languages.[4] James Gosling, who created the Java programming language in 1994, called it an 'imitation' of that language.[5] The most recent version is C# 3.0, which was released in conjunction with the .NET Framework 3.5 in 2007. The next proposed version, 4.0, is in development."

That is certainly a mouthful.

Why should you care about C#?

From the viewpoint of students in this course, here is some of what C# really is and why you should care about C#.

Complex IDE

To begin with, C# was designed to run in a large complex IDE. For example, using the Visual C# IDE to create the simple Hello World console program shown in [Listing 1](#) generates about thirteen different files in about ten different folders.

Note:

Listing 1 . Hello World in C#.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Hello01{
    class Hello01{
        static void Main(string[] args){
            Console.WriteLine("Hello C# World");
            //Press any key to dismiss console
screen.
            Console.ReadKey();
        } //end Main
    } //end class
} //end namespace
```

Is there a simpler way?

While it is possible to develop simple C# programs outside the IDE (see [Baldwin's C# programming tutorials](#)), there are probably less than a few hundred people worldwide who do it that way.

The transition from the relatively simple IDE used in the prerequisite course to the very complex Visual C# IDE may be a stretch for some students.

Fully object oriented

C# is an object-oriented programming language. Unlike C++, it is not possible to develop C# projects without taking the object-oriented nature of C# into account.

Perhaps more important, it is not possible to develop XNA programs without using some of the more complex aspects of OOP, such as the overriding of virtual methods.

Massive and complex documentation package

The object-oriented nature of C# becomes very apparent when attempting to locate something in the massive XNA documentation package. The documentation is written from an object-oriented viewpoint. For example, see the documentation for the **Microsoft.Xna.Framework.Graphics** namespace [here](#) and see the documentation for the **SpriteBatch** class [here](#).

Indirection

C# makes extensive use of indirection. While the indirection scheme used in C# is not as complicated as C++ pointers, for someone new to indirection, understanding the concept can sometimes be a challenging task.

C# is the language of XNA

C# is the programming language of the Microsoft XNA Game Studio. To write XNA programs, you must know how to write C# programs.

That is why you should care about C#. I won't try to turn you into a C# programming expert in this course. We have other courses at the college that concentrate solely on C# programming. Instead, I will simply try to help you learn enough about C# programming that you can do a credible job of writing programs using the XNA Game Studio.

Run the program

Click [here](#) to download a zip file containing my version of the program. Extract the folder named **ConsoleApplication1** from the zip file and save it somewhere on your disk. Start **Visual C# 2010 Express** and select **Open Project...** from the **File** menu. Navigate to the project folder and select the file with the extension of **.sln** . This should cause the project to open and be ready to run or debug as described in the earlier module titled [Getting Started](#) .

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Xna0102-What is C# and Why Should You Care
- File: Xna0102.htm
- Published: 02/24/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Xna0104-What is OOP and Why Should You Care?

Learn about object-oriented programming in general. Also learn about the structure and syntax of an object-oriented C# program by taking a simple program apart and examining the elements of the program.

Revised: Wed May 04 17:44:09 CDT 2016

This page is part of a Book titled [XNA Game Studio](#).

Table of Contents

- [Table of Contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [What is OOP?](#)
 - [What is object-oriented programming.\(OOP\)?](#)
 - [My answer](#)
 - [An anecdotal description](#)
 - [Atomic and non-atomic objects](#)
 - [Your job - assemble the objects](#)
 - [Objects working together](#)
 - [Creating a model](#)
 - [Objects must be designed and manufactured](#)
 - [A class is analogous to manufacturing drawings](#)
 - [A large library of classes](#)
 - [Classes of your own design](#)
- [Why should you care about OOP?](#)
 - [The XNA documentation package is huge](#)
 - [The good news](#)

- [Three important concepts](#)
 - [Encapsulation example](#)
 - [Only the interface is exposed](#)
 - [How is it implemented?](#)
 - [Expose the interface and hide the implementation](#)
 - [Inheritance example](#)
 - [OOP lingo](#)
 - [Extending the Game class](#)
 - [Reuse, don't reinvent](#)
 - [Polymorphism example](#)
 - [Select Drive to go forward](#)
- [Object-oriented programming vocabulary](#)
- [Discussion and sample code](#)
 - [You can learn a lot...](#)
 - [A new class definition](#)
 - [The Main method](#)
 - [Methods versus functions](#)
 - [Definition of the Main method](#)
 - [The WriteLine method](#)
 - [The Console class](#)
 - [The System namespace](#)
 - [Members of the Console class](#)
 - [The dot operator](#)
 - [Namespaces](#)
 - [The "using" declaration](#)
 - [Not much help in this case](#)
 - [Defining your own namespace](#)
 - [The project file structure](#)
 - [The Hello01 namespace](#)
 - [The call to the ReadKey method](#)

- [Run the program](#)
- [Run my program](#)
- [Summary](#)
- [Miscellaneous](#)

Preface

This module is one in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

An earlier module titled [Getting Started](#) provided information on how to get started programming with Microsoft's XNA Game Studio.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Project file structure.

Listings

- [Listing 1](#). Extending the Game class.
- [Listing 2](#). Hello World in C#.

What is OOP?

What is object-oriented programming (OOP)?

If you Google this question, you will get hundreds of answers. Here is one of those answers.

According to Wikipedia,

Note: "Object-oriented programming (OOP) is a programming paradigm that uses "objects" - data structures consisting of data fields and methods together with their interactions - to design applications and computer programs. Programming techniques may include features such as information hiding, data abstraction, encapsulation, modularity, polymorphism, and inheritance."

My answer

Here is my answer along with an anecdotal description. Unlike earlier programming styles, object-oriented programming is a programming style that mimics the way most people think and work.

An OOP solution to a problem should resemble the problem, and observers of the solution should be able to recognize the problem without necessarily knowing about it in advance.

For example, an OO program that deals with banking transactions should be recognizable on the basis of the objects that it uses, such as deposit objects, withdrawal objects, account objects, etc.

An anecdotal description

If you have ever assembled a playscape in your back yard, this should sound familiar to you.

When you opened the large boxes containing the playscape, hundreds of objects spilled onto the ground. Those objects may have consisted of braces, chains, swing seats, slides, screws, nuts, bolts, washers, climbing ropes, ladder rungs, and other assorted objects.

Atomic and non-atomic objects

I will refer to (*most of*) the kinds of object that I have described in the above list as atomic objects. What I mean by that is that they can't be easily subdivided into smaller objects without destroying their functionality.

If you were lucky, some of the objects in the box may not have been atomic objects. Instead they may have been pre-assembled arrangements of atomic objects such as an assembly of seats and braces representing a object on which two children can swing together.

Your job - assemble the objects

Your job was to assemble those hundreds of atomic and non-atomic objects into a final object which you proudly referred to as "*The Playscape*."

Objects working together

It has been said that a successful object-oriented program consists of a bunch of cooperating software objects working together to achieve a specified behavior.

The overall behavior of the program is the combination of behaviors of the individual objects. For example, some objects may acquire input data, other objects may compute and produce output data, while other objects may display the output data.

It could also be said that a playscape consists of a bunch of hardware objects working together to achieve a specified behavior. The overall behavior of the playscape is the combination of behaviors of the individual objects. For example, the behavior of some of the braces is to stand strong and not bend, while the behavior of a swing chain is to be flexible and move in a prescribed way.

Creating a model

One of the tasks often faced by an object-oriented programmer is to assemble software objects into a model that represents something that exists in the real world. As a very visual example, you might be asked to create an advertising web page showing an animated software model of the playscape that you assembled in your back yard. With the playscape, you were simply required to assemble the existing objects. However, in the object-oriented programming world, you must do more than just assemble objects.

Objects must be designed and manufactured

Getting back to the playscape, each of the objects for the playscape was manufactured before being shipped to you. Even before that, each object was designed by someone and a set of manufacturing drawings was probably created so that the object could be mass produced.

A class is analogous to manufacturing drawings

In OOP, there is a direct analogy to the manufacturing drawings of the hardware world. We call it a *class*. A class documents the specifications for the construction of a particular type of software object. For example, there is probably a set of classes that describe the specifications for each of the button objects and menu objects at the top of the browser in which you are currently viewing this module.

A large library of classes

As an object-oriented programmer, you will typically have access to a large library of existing classes from which you can construct different types of software objects, such as buttons, sliders, etc. For example, you will find links to the various XNA classes [here](#).

Classes of your own design

In addition, you will often need to design and define new classes from which you can construct new types of objects.

Why should you care about OOP?

You need to care about OOP because the language of XNA is C# and C# is an object-oriented programming language. It is not possible to write a C# program without dealing with the object-oriented nature of the language.

For most humans, it is also not possible to write credible XNA programs without frequent reference to the XNA documentation. Although the use of the XNA framework will shield you from some of the difficulties of object-oriented programming, you will still have both feet in the OOP sandbox as soon as you consult the documentation.

The XNA documentation package is huge

For example, the [XNA Framework Class Library](#) lists about a dozen namespaces (*over and above the namespaces in the standard C# library*) .

One of those namespaces is named **Microsoft.Xna.Framework.Graphics** . That namespace lists about 175 different classes.

One of those classes is the **SpriteBatch** class. The **SpriteBatch** class lists several members including one constructor, four public properties, nine

public methods, two protected methods, and one event.

The **SpriteBatch** class is probably one of the most commonly used classes in the XNA framework so you will need to be able to understand the documentation for all or at least most of the members of that class.

The good news

The good news is that the documentation also provides numerous code samples and explanatory notes to help you use the documentation to write your code correctly.

Three important concepts

Any object-oriented language must support three very important concepts:

- Encapsulation
- Inheritance
- Polymorphism

We use these three concepts extensively as we attempt to model the real-world problems that we are trying to solve with our object-oriented programs. I will provide brief descriptions of these concepts in this module and then explain each concept in detail in future modules.

Encapsulation example

Consider the steering mechanism of a car as a real-world example of encapsulation. During the past eighty years or so, the steering mechanism for the automobile has evolved into an object in the OOP sense.

Only the interface is exposed

In particular, most of us know how to use the steering mechanism of an automobile without having any idea whatsoever how it is implemented. All most of us care about is the interface, which we refer to as a steering wheel. We know that if we turn the steering wheel clockwise, the car will turn to the right, and if we turn it counterclockwise, the car will turn to the left.

How is it implemented?

Most of us don't know, and don't really care, how the steering mechanism is actually implemented "under the hood." In fact, there are probably a number of different implementations for various brands and models of automobiles. Regardless of the brand and model, however, the human interface is pretty much the same. Clockwise turns to the right, counterclockwise turns to the left.

Expose the interface and hide the implementation

As in the steering mechanism for a car, a common approach in OOP is to *"hide the implementation"* and to *"expose the interface"* through encapsulation.

Inheritance example

Another important aspect of OOP is inheritance. Let's form an analogy with the teenager who is building a hotrod. That teenager doesn't normally start with a large chunk of steel and carve an engine out of it. Rather, the teenager will usually start with an existing engine and make improvements to it.

OOP lingo

In OOP lingo, that teenager *extends* the existing engine, *derives* from the existing engine, *inherits* from the existing engine, or *subclasses* the existing engine (*depending on which author is describing the process*) .

Just like in "*souping up*" an engine for a hotrod, a very common practice in OOP is to create new improved classes and objects by extending existing class definitions.

Extending the Game class

When you use Visual C# to create a new XNA game project, the IDE creates skeleton code for a new class. As you will learn later in this course, it is then up to you to put some meat on the skeleton and turn it into a game. The first executable statement in the skeleton code is shown in [Listing 1](#).

Note:

Listing 1 . Extending the Game class.

```
public class Game1 : Microsoft.Xna.Framework.Game
```

The code in [Listing 1](#) is the first executable statement in the definition of a new class named **Game1** (*or whatever name you give the new project*) . The definition of the new class extends an existing class named **Game** , which resides in the **Microsoft.Xna.Framework** namespace. The existing class named **Game** , is the main controller for your new game. I will have much more to say about the **Game** class in a future module.

Reuse, don't reinvent

One of the major arguments in favor of OOP is that it provides a formal mechanism that encourages the reuse of existing programming elements. One of the mottos of OOP is "*reuse, don't reinvent.*"

Polymorphism example

A third important aspect of OOP is polymorphism. This is a Greek word meaning something like *one name, many forms* . This is a little more difficult to explain in non-programming terminology. However, we will stretch our imagination a little and say that polymorphism is somewhat akin to the automatic transmission in your car. In my Honda, for example, the automatic transmission has four different methods or functions known collectively as Drive (*in addition to the functions of Reverse, Park, and Neutral*) .

Select Drive to go forward

As an operator of the automobile, I simply select Drive (*meaning go forward*) . Depending on various conditions at runtime, the automatic transmission system decides which version of the Drive function to use in every specific situation. The specific version of the function that is used is based on the current conditions (*speed, incline, etc.*) . This is somewhat analogous to what we will refer to in a subsequent tutorial module as runtime polymorphism.

Object-oriented programming vocabulary

OOP involves a whole new vocabulary (*or jargon*) which is different from or supplemental to the vocabulary of procedural programming.

For example the object-oriented programmer defines an *abstract data type* by encapsulating its implementation and its interface in a class.

One or more *instances* of the class can then be created or *instantiated* .

An *instance* of a class is known as an *object* .

Every object has *state* and *behavior* where the state is determined by the current values stored in the object's *instance variables* and the behavior is determined by the *instance methods* belonging to the object.

Inherited abstract data types are *derived* classes or *subclasses* of *base* classes or *super* classes. We *extend* super classes to create subclasses.

Within the program the code *instantiates* objects (*creates instances of classes*) and *sends messages* to the objects by calling the class's *methods* (or *member functions*).

If a program is "object oriented", it uses *encapsulation* , *inheritance* , and *polymorphism* . It defines abstract data types, encapsulates those abstract data types into classes, instantiates objects from the classes, and sends messages to the objects.

The members of a class fall generally into one of the following categories:

- Constructors
- Properties
- Methods
- Events

The individual members of a class can be *public* , *private* , or *protected* .

To make things even more confusing, almost every item or action used in the OOP jargon has evolved to be described by several different terms. For example, we can cause an object to *change its state* by sending it a message, calling its methods, or calling its member functions. The term being used often depends on the author who wrote the specific book that you happen to be reading at the time.

Hopefully most of this terminology will become clear as we pursue these modules

Discussion and sample code

I usually try to provide some code in each module that you can compile and execute. [Listing 2](#) contains the C# code for a simple program that will display the words "Hello C# World" on the system console screen when you compile and run it. *(The system console screen will probably appear as a black window when you run the program.)*

Note:

Listing 2 . Hello World in C#.

```
//File Program.cs
using System;
namespace Hello01{

    class Hello01{

        static void Main(string[] args){
            Console.WriteLine("Hello C# World");
            //Press any key to dismiss console
screen.
            Console.ReadKey();
        }//end Main

    }//end class definition
}//end namespace
```

You can learn a lot...

You may be surprised at how much you can learn about the structure and syntax of an object-oriented program in C# by taking this simple program apart and examining the elements of the program. Note, however, that this program is strictly class based. It **does not instantiate any objects** .

A new class definition

The central block of code beginning with the word **class** defines a new class named **Hello01** .

The Main method

You probably learned that every C++ program requires a function named **main** . Execution of the C++ program begins and ends in the **main** function.

The same is true in C#. Every C# program requires a method named **Main** (*note the upper-case "M"*) . However, unlike in C++, the **Main** method in C# must be defined inside of a class definition.

Methods versus functions

Methods in C# are analogous to the functions defined inside a class in C++. However, they are called methods instead of functions in C#.

Definition of the Main method

The **Main** method in [Listing 2](#) begins with the keyword **static** and ends with the comment "end Main".

The WriteLine method

As you may have guessed already, the call to the **WriteLine** method inside the **Main** method causes the **WriteLine** method's argument to be displayed on the system console.

The Console class

[Console](#) is the name of a *static* class in the [System](#) namespace. The significance of the class being static is that *"You do not need to declare an instance of a static class in order to access its members."* Therefore, as I mentioned earlier, this program doesn't purposely instantiate any new objects.

According to the [documentation](#), the **Console** class *"Represents the standard input, output, and error streams for console applications. This class cannot be inherited."*

The System namespace

The [System](#) namespace actually belongs to the Microsoft .NET framework and the classes in the namespace are available for use by several different programming languages that also belong to the framework such as C#, VB.NET, etc. (See the online *.NET Framework Class Library* [here](#).)

Members of the Console class

The **Console** class has many members, about nineteen of which are overloaded versions of the method named [WriteLine](#). (Method overloading means that the same method name can be used two or more times in the same scope as long as the argument list differs from one version to the next.)

The version of the **WriteLine** method that is called in [Listing 2](#) requires a **string** object as an incoming parameter. According to the documentation, the behavior of this version of the method is:

Note:

"Writes the specified string value, followed by the current line terminator, to the standard output stream."

The dot operator

Note the period that joins the name of the class and the name of the method in **Console.WriteLine** . When used in this way, the period is often referred to as the *dot operator* . The dot operator is used in a variety of similar but different ways in C#. In this case, it tells the compiler to look in the **Console** class for a method named **WriteLine** that requires an incoming parameter of type **string** . *(It is actually the type of parameter being passed to the method, **string** , that specifies which version of the **WriteLine** method will be executed.)*

Namespaces

Briefly, namespaces provide a way to partition the class library and the new classes defined in a program so that it is possible to reference two classes with the same name in the same scope as long as they are in different namespaces. Therefore, whenever you reference a class name in your code, you must tell the compiler which namespace it resides in. This can be done in two ways. One way is to write the namespace in front of the class name joined by the dot **operator as in**

System.Console

or

Microsoft.Xna.Framework.Graphics.SpriteBatch

The "using" declaration

The second way to tell the compiler which namespace the class resides in is through the use of a *"using"* declaration.

As long as it won't create name conflicts, you can tell the compiler that you are *"using"* the namespace as shown by the first line following the initial comment in [Listing 2](#). Then you don't need to refer to the namespace when you reference a class belonging to that namespace in your code.

Not much help in this case

In this case, since the **Console** class was referenced only once, it would have been simpler to join the class name with the namespace name using the dot operator and to omit the *"using"* declaration. However, in those cases where the class name is referenced more than once in the code, it is simpler to declare the namespace at the beginning to avoid having to type it more than once. This is particularly true in the case of the **SpriteBatch** class shown above where the namespace name contains several levels separated by periods.

Defining your own namespace

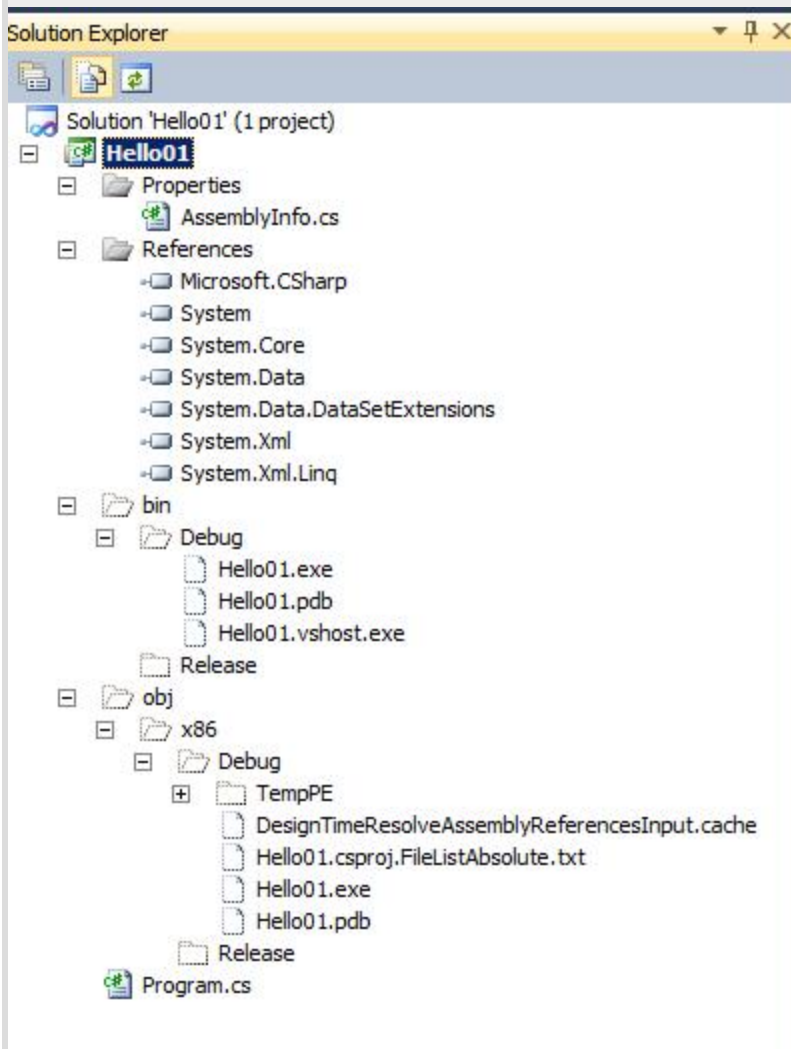
When you define a new class, you should always define the namespace in which it resides. (*I believe this is a requirement when developing projects using Visual C#.*)

The project file structure

[Figure 1](#) shows the project file structure for the **Hello01** project. ([Figure 1](#) was captured from the **Solution Explorer** window in the Visual C# IDE.)

Note:

Figure 1 . Project file structure.



The class definition shown in [Listing 2](#) is contained in the file named **Program.cs** at the bottom of the tree in [Figure 1](#). Note that this file is contained in the folder named **Hello01**. Also note that the folder named **Hello01** is a child of another folder named **Hello01** at the top of the project tree. (There are two folders named **Hello01** in the project tree.)

The Hello01 namespace

The second line following the initial comment in [Listing 2](#) defines the namespace in which the new class resides. Note that it is the name of the folder in which the file resides. Note also that it is specified relative to the top of the project tree.

If the folder containing the class happened to be more than one level down in the project tree, as is the case of the **Graphics** folder that contains the **SpriteBatch** class, it would be necessary to use periods to trace down the tree.

The call to the ReadKey method

[Listing 2](#) also contains a call to a method named **ReadKey**, which also belongs to the **Console** class. The purpose of this call is to cause the execution of the program to block and wait until the user presses any key. Otherwise, the console window would appear on the computer screen momentarily and then disappear almost as soon as it appears.

Run the program

I encourage you to copy the code from [Listing 2](#). Use that code to create a Visual C# **Console Application** project. Build and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Run my program

Click [here](#) to download a zip file containing my version of the program. Extract the folder named **Hello01** from the zip file and save it somewhere on your disk. Start **Visual C# 2010 Express** and select **Open Project...** from the **File** menu. Navigate to the project folder and select the file with the extension of **.sln**. This should cause the project to open and be ready to run or debug as described in the earlier module titled [Getting Started](#).

Summary

You learned something about object-oriented programming in general in this module. You also learned quite a bit about the structure and syntax of an object-oriented C# program by taking a simple program apart and examining the elements of the program.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Xna0104-What is OOP and Why Should You Care?
- File: Xna0104.htm
- Published: 02/24/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Xna0106-Encapsulation in C#

Learn about encapsulation, abstraction, abstract data types, information hiding, class member access control, instance variables, local variables, class variables, reference variables, primitive variables, how C# identifies properties, and how to set and get properties in C#. Also learn about public accessor methods, public manipulator methods, data validation by property setters and public access methods, and a few other things along the way.

Revised: Thu May 05 12:44:19 CDT 2016

This page is part of a Book titled [XNA Game Studio](#).

Table of Contents

- [Table of Contents](#)
- [Preface](#)
 - [The three main characteristics of an object-oriented program](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [Abstraction](#)
 - [How does abstraction relate to encapsulation?](#)
 - [Some analogies](#)
 - [A new type](#)
 - [Already known to the compiler](#)
 - [Not known to the compiler](#)
 - [Define data representation and behavior in a class](#)
 - [Create instances of the new type](#)
 - [Objects have state and behavior](#)
 - [The state and behavior of a GUI Button object](#)

- [A C# class named Button](#)
 - [The state of Button objects](#)
 - [The behavior of a Button object](#)
- [Encapsulation](#)
 - [Expose the interface and hide the implementation](#)
 - [Should be able to change the implementation later](#)
 - [Class member access control](#)
 - [Five levels of access control](#)
 - [A different interpretation](#)
 - [What is an assembly?](#)
 - [Public, private, and protected](#)
 - [A public user interface](#)
 - [A set Accessor method](#)
 - [A get Accessor method](#)
 - [Not a good design by default](#)
 - [Not bound to the implementation](#)
- [Preview](#)
- [Discussion and sample code](#)
 - [Will explain in fragments](#)
 - [Two classes](#)
 - [Will switch between classes](#)
 - [The declarations](#)
 - [Beginning of the class named Props01](#)
 - [Access a public instance variable in the object](#)
 - [Store a string value in the object's instance variable](#)
 - [Get and display the value in the object's instance variable](#)
 - [Beginning of the class named TargetClass](#)
 - [Declare a public instance variable](#)

- [Different kinds of variables](#)
 - [Instance variables](#)
 - [The public access modifier](#)
 - [The private access modifier](#)
 - [The protected access modifier](#)
 - [Local variables](#)
 - [Class variables](#)
 - [More bad programming practice](#)
 - [Reference variables](#)
 - [Primitive variables](#)
 - [An analogy](#)
- [Call the object's public accessor methods](#)
 - [The setColor method](#)
 - [The getColor method](#)
- [Would be a property in Java](#)
- [The definition of setColor and getColor in TargetClass](#)
 - [Typical method definitions](#)
 - [Data validation](#)
 - [What do you do with invalid data](#)
 - [Program output](#)
 - [Set, get, and display a property value with a valid value](#)
 - [Looks can be deceiving](#)
 - [A set Accessor method](#)
 - [A get Accessor method](#)
 - [The set Accessor and get Accessor methods for the property named height](#)

- [A hidden parameter named value](#)
- [Validating code](#)
- [A read-only property](#)
- [Little more to say about this](#)
- [Call manipulator method and display results](#)
 - [A manipulator method named doubleHeight](#)
- [Set and get the property with an invalid value](#)
- [Pause until the user presses any key](#)
- [The end of the program](#)
- [Run the program](#)
- [Run my program](#)
- [Summary](#)
- [Miscellaneous](#)
- [Complete program listing](#)

Preface

This module is one in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

An earlier module titled [Getting Started](#) provided information on how to get started programming with Microsoft's XNA Game Studio.

The three main characteristics of an object-oriented program

Object-oriented programs exhibit three main characteristics:

- Encapsulation
- Inheritance
- Polymorphism

In this and the next two modules, I will explain and illustrate those three characteristics plus some related topics.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Output from the program named Props01.

Listings

- [Listing 1](#). Beginning of the class named Props01.
- [Listing 2](#). Beginning of the class named TargetClass.
- [Listing 3](#). Call the object's public accessor methods.
- [Listing 4](#). Definition of setColor and getColor methods in TargetClass.
- [Listing 5](#). Set, get, and display a property value with a valid value.
- [Listing 6](#). The set Accessor and get Accessor methods for the property named height.
- [Listing 7](#). Call manipulator method and display results.
- [Listing 8](#). A simple manipulator method named doubleHeight.
- [Listing 9](#). Set and get the property named height with an invalid value.
- [Listing 10](#). Pause until the user presses any key.
- [Listing 11](#). The program named Props01.

General background information

In addition to the three explicit characteristics of encapsulation, inheritance, and polymorphism, an object-oriented program also has an implicit characteristic of *abstraction* .

Abstraction

Abstraction is the process by which we specify a new data type, often referred to as an abstract data type or ADT. This includes a specification of the type's data representation and its behavior. In particular,

- What kind of data can be stored in an entity of the new type?
- What are all the ways that the data can be manipulated?

How does abstraction relate to encapsulation?

Encapsulation is the process of gathering an ADT's data representation and behavior into one encapsulated entity. In other words, encapsulation converts from the *abstract* to the *concrete*.

Some analogies

You might think of this as being similar to converting an idea for an invention into a set of blueprints from which it can be built, or converting a set of written specifications for a widget into a set of drawings that can be used by the machine shop to build the widget.

Automotive engineers encapsulated the specifications for the steering mechanism of my car into a set of manufacturing drawings. Then manufacturing personnel used those drawings to produce an object where they exposed the interface (steering wheel) and hid the implementation (levers, bolts, etc.).

In all likelihood, the steering mechanism contains a number of other more-specialized embedded objects, each of which has state and behavior and each of which also has an interface and an implementation.

The interfaces for those embedded objects aren't exposed to me, but they are exposed to the other parts of the steering mechanism that use them.

A new type

For our purposes, an abstract data type is a new type (not intrinsic to the C# language). It is not one of the primitive data types that are built into the programming language such as **sbyte** , **short** , **int** , **long** , **float** , **double** , etc.

Already known to the compiler

The distinction in the previous paragraph is very important. The data representation and behavior of the intrinsic or primitive types is already known to the compiler and cannot normally be modified by the programmer.

Not known to the compiler

The representation and behavior of an abstract type is not known to the compiler until it is defined by the programmer and presented to the compiler in an appropriate manner.

Define data representation and behavior in a class

C# programmers define the data representation and the behavior of a new type (present the specification to the compiler) using the keyword **class** . In other words, the keyword **class** is used to convert the specification of a new type into something that the compiler can work with; a set of plans as it were. To define a class is to go from the abstract to the concrete.

Create instances of the new type

Once the new type (class) is defined, one or more objects of that type can be brought into being (instantiated, caused to occupy memory).

Objects have state and behavior

Once instantiated, the object is said to have *state* and *behavior*. The state of an object is determined by the current values of the data that it contains. The behavior of an object is determined by its methods.

The state and behavior of a GUI Button object

For example, if we think of a GUI **Button** as an object, it is fairly easy to visualize the object's state and behavior.

A GUI **Button** can usually manifest many different states based on size, position, depressed image, not depressed image, label, etc. Each of these states is determined by data stored in the instance variables of the **Button** object at any given point in time. (The combination of one or more instance variables that determine a particular state is often referred to as a *property* of the object.)

Similarly, it is not too difficult to visualize the behavior of a GUI Button. When you click it with the mouse, some specific action usually occurs.

A C# class named Button

If you dig deeply enough into the C# [class library](#), you will find that there is a class named [Button](#) in the **System.Windows.Forms** namespace. Each individual **Button** object in a **C# Windows Forms Application** is an instance of the C# class named **Button**. A **Button** object has a single constructor, dozens of methods, dozens of properties, and dozens of events.

The state of Button objects

Each **Button** object has instance variables, which it does not share with other **Button** objects. The values of the instance variables define the state of the button at any given time. Other **Button** objects in the same scope can

have different values in their instance variables. Hence every **Button** object can have a different state.

The behavior of a Button object

Each Button object also has certain fundamental behaviors such as responding to a mouse **Click** event or responding to a **GotFocus** event.

The C# programmer has control over the code that is executed in response to the event. However, the C# programmer has no control over the fact that a **Button** object will respond to such an event. The fact that a **Button** will respond to certain event types is an inherent part of the type specification for the **Button** class and can only be modified by modifying the source code for the **Button** class.

Encapsulation

If abstraction is the design or specification of a new type, then encapsulation is its definition and implementation.

A programmer defines the data representation and the behavior of an abstract data type into a class, thereby defining its implementation and its interface. That data representation and behavior is then encapsulated in objects that are instantiated from the class.

Expose the interface and hide the implementation

According to good object-oriented programming practice, an encapsulated design usually exposes the interface and hides the implementation. This is accomplished in different ways with different languages.

Just as most of us don't usually need to care about how the steering mechanism of a car is implemented, a user of a class should not need to care about the details of implementation for that class.

The user of the class (the using programmer) should only need to care that it works as advertised. Of course this assumes that the user of the class has access to good documentation describing the interface and the behavior of objects instantiated from the class.

Should be able to change the implementation later

For a properly designed class, the class designer should be able to come back later and change the implementation, perhaps changing the type of data structure used to store data in the object, and the using programs should not be affected by the change.

Class member access control

Object-oriented programming languages usually provide the ability to control access to the members of a class. For example, C#, C++ and Java all use the keywords **public**, **private**, and **protected** to control access to the individual members of a class. In addition, Java adds a fourth level of access control, which is called **package-private**. C# adds two levels of access control that are not included in Java or C++ (see the next section).

Five levels of access control

According to the C# [specifications](#), "Each member of a class has an associated accessibility, which controls the regions of program text that are able to access the member." There are five levels of access control in C#:

- **public** - Access not limited
- **protected** - Access limited to this class or classes derived from this class
- **internal** - Access limited to this program
- **protected internal** - Access limited to this program or classes derived from this class
- **private** - Access limited to this class

A different interpretation

Another online [source](#) provides a different interpretation for two of the access levels:

- The **internal** modifier declares that a member is known throughout all files in an assembly, but unknown outside that assembly.
- The **protected internal** access level can be given only to class members.
- A member declared with **protected internal** access is accessible within its own assembly or to derived types.

What is an assembly?

You can learn more about an assembly [here](#). Frankly, I'm not absolutely certain at this time how to interpret the access levels of *internal* and *protected internal*. However, I believe that an assembly in C# is similar to a package in Java, and if so, then I do know how to interpret these two access levels.

Public, private, and protected

To a first approximation, you can probably guess what **public** and **private** mean. **Public** members are accessible by all code that has access to an object of the class. **Private** members are accessible only by members belonging to the class.

The **protected** keyword is used to provide inherited classes with special access to the members of their base classes.

A public user interface

In general, the user interface for a class consists of the **public** methods.

Note: The variables in a class can also be declared **public** but this is generally considered to be bad programming practice unless they are actually constants.

For a properly designed class, the class user stores, reads, and modifies values in the object's data by calling the **public** methods on a specific instance (object) of the class. (This is sometimes referred to as sending a message to the object asking it to change its state).

Normally, if the class is properly designed and the implementation is hidden, the user cannot modify the values contained in the private instance variables of the object without going through the prescribed public methods in the interface.

A set Accessor method

C# has a special form of method, often called a *set Accessor* method. The use of this type of method makes it appear that an assignment is being made to store a value in a private instance variable belonging to an object when in fact, the assignment operation is automatically converted to a call to a *set Accessor* method. I discuss this more fully in my earlier tutorial titled [Learning C# and OOP, Properties, Part 1](#). I will also show an example of a *set Accessor* method later.

A get Accessor method

C# also has a special form of method often called a *get Accessor* method that operates like a *set Accessor* method but in the reverse direction. A *get Accessor* method makes it appear that the value of a private instance variable can be obtained by referencing the name of the object joined to the name of the variable. In fact, that reference is automatically converted to a call to a *get Accessor* method.

Not a good design by default

An object-oriented design is not a good design by default. In an attempt to produce good designs, experienced object-oriented programmers generally agree on certain design standards for classes. For example, the data members (instance variables) are usually private unless they are constants. The user interface usually consists only of public methods and includes few if any data members.

Of course, there are exceptions to every rule. One exception to this general rule is that data members that are intended to be used as symbolic constants are made public and defined in such a way that their values cannot be modified.

The methods in the interface should control access to, or provide a pathway to the private instance variables.

Not bound to the implementation

The interface should be generic in that it is not bound to any particular implementation. Hence, the class author should be able to change the implementation without affecting the using programs so long as the interface doesn't change.

In practice, this means that the signatures of the interface methods should not change, and that the interface methods and their arguments should continue to have the same meaning even if the author of the class changes the internal implementation.

Preview

I will present and explain a simple C# console program named **Probs01** that illustrates encapsulation and C# properties in the remainder of this module. The output from the program is shown in [Figure 1](#).

Note:

Figure 1 . Output from the program named Props01.

```
text: Quit  
color: Red  
color: Bad color  
height: 20  
double height: 40  
height: 0
```

Discussion and sample code

Will explain in fragments

I will explain the code in this program in fragments. A complete listing of the program is provided in [Listing 11](#) near the end of the module.

Two classes

The program contains two separate class definitions. One class, named **TargetClass** , illustrates encapsulation and properties. The other class named **Props01** instantiates an object of **TargetClass** and exercises its interface. For simplicity, both classes were defined in the same physical file, but that is not a requirement.

Will switch between classes

In an attempt to help you understand how the program works, I will switch back and forth between the two classes showing the cause and effect relationship between the code in one class and the code in the other class.

Note:

Listing 1 . Beginning of the class named Props01.

```
using System;
namespace Props01{

    class Props01{

        static void Main(string[] args){
            TargetClass obj = new TargetClass();

            //Access a public instance variable
            obj.text = "Quit";
            Console.WriteLine("text: " + obj.text);
        }
    }
}
```

The declarations

The first two statements in [Listing 1](#) apply equally to both classes. The first statement "uses" the namespace named **System** . This eliminates the requirement to qualify every reference to the **Console** class with the name of the namespace containing the **Console** class.

The second statement in [Listing 1](#) establishes the namespace for the new program code, which is actually the name of the folder containing the source code file.

Beginning of the class named Props01

The class definition for the class named **Props01** begins with the keyword **class** shown in [Listing 1](#).

As you will see when we get to the end of the explanation, the only thing that is contained in this class is the **Main** method. The required signature

for the **Main** method is shown in [Listing 1](#). I'm not going to explain all of the ramifications for the syntax of the method signature. At this point, I will simply tell you to memorize the syntax.

The first statement in the **Main** method uses the **new** operator to instantiate an object of the class named **TargetClass** and save a reference to that object in a local reference variable named **obj**. This reference will be used later to access the object.

Access a public instance variable in the object

The last two statements in [Listing 1](#) access a public instance variable belonging to the object. The first of the two statements assigns the string value "Quit" to the variable. The second statement retrieves and displays that value.

Store a string value in the object's instance variable

Note the syntax of the first of these two statements. The value of the reference variable named **obj** is joined to the name of the object's public instance variable. (The name of the instance variable is **text**.) The assignment operator is used to store a string value in that instance variable.

You can think of this operation as involving the following steps:

1. Get the object's reference from the variable named **obj**.
2. Use that reference to locate the object in memory.
3. Knock on the object's door and ask for access to the instance variable named **text**. (Access will be granted because the instance variable is declared **public** as you will see shortly.)
4. Store the string value "Quit" in the instance variable using an assignment operator.

Get and display the value in the object's instance variable

The last statement does essentially the same thing in reverse.

1. Get and use the object's reference to gain access to the object's public instance variable.
2. Get the value stored in that instance variable.
3. Concatenate that value to the literal string value "text".
4. Pass the concatenated string to the **WriteLine** method of the static **Console** class to have it displayed on the standard output device (the black screen).

This produces the first line of output shown in [Figure 1](#).

Beginning of the class named TargetClass

The class definition for the class named **TargetClass** begins in [Listing 2](#).

Note:

Listing 2 . Beginning of the class named TargetClass.

```
public class TargetClass{  
    public string text;
```

Declare a public instance variable

The code in the class begins by declaring a public instance variable. This is considered to be bad programming practice in most quarters unless the public variable is actually a constant, (which it is not).

This is the variable that is accessed by the last two statements in [Listing 1](#).

In most cases, instance variables should be declared private and should be made accessible through public accessor methods or public set and get methods. You will see examples of public access, set, and get methods later.

Different kinds of variables

I will generally refer to three kinds of variables:

1. **Instance variables** - declared inside a class but outside of a method or constructor. (All variables in C# must be declared inside a class. Unlike C++, there are no global variables or global functions in C#.)
2. **Local variables** - declared inside a method or constructor.
3. **Class variable** - declared inside a static class.

Any of these can be further qualified as follows:

1. **Reference variable** - contains a reference to an object or contains null.
2. **Primitive variable** - contains a value of a primitive type (int, float, double, etc.).

Instance variables

An instance variable belongs to a specific object. The lifetime of an instance variable is the same as the lifetime of the object to which it belongs. The scope of an instance variable depends on its access modifier such as **public** , **private** , or **protected** .

The public access modifier

A **public** instance variable can be accessed by any code in any method in any object that can obtain a reference to the object to which the variable belongs. However, you must first gain access to the object in order to gain access to the variable. Normally you gain access to the object using a

reference to the object and then gain access to a member of the object such as a variable or method. This is *indirection* .

The private access modifier

A **private** instance variable can be accessed by any code in any method that is defined in the class of the object to which the variable belongs.

The protected access modifier

A **protected** instance variable can be accessed by the same methods that can access a **private** instance variable plus methods in subclasses of the class of the object to which the instance variable belongs.

I'm not going to try to explain the scope of **internal** and **protected internal** instance variables for the reasons that I discussed earlier.

Local variables

A local variable is declared inside a method or constructor. The lifetime of a local variable is limited to the time that control remains within the block of code in which the variable is declared. The scope of the variable is limited to the code block in which it is declared and then only to the statements following the declaration statement.

Class variables

A class variable belongs to a static class. I believe that the lifetime of a class variable is the same as the lifetime of the program in which the class is loaded.

It is not necessary to instantiate an object of the static class in order to access the variable. Assuming that you have access rights to the variable,

you can access it simply by joining the name of the class to the name of the variable using the dot operator.

More bad programming practice

I personally consider it bad programming practice to use class variables in most cases unless the variables are actually constants. However, there are a few situations where you have no choice but to use a non-constant class variable.

Reference variables

A reference variable contains a reference to an object or contains null. You typically use the value stored in a reference variable to locate an object in memory. Once you locate the object, you typically use the dot operator along with the name of a variable or a method to ask the object to do something. This is called *indirection*.

Primitive variables

Primitive variables contain primitive values. No indirection is required to access the primitive value stored in a primitive variable.

An analogy

An analogy that I often use to explain the difference between a reference variable and a primitive variable goes as follows.

A primitive variable is analogous to your wallet.

If you get robbed and the robber takes your wallet, he has your money because the wallet contains your money just like a primitive variable contains a primitive value.

A reference variable is analogous to your check book.

If the robber takes your checkbook, he doesn't have your money -- not yet anyway. The checkbook doesn't contain your money. Instead, it contains a reference to your bank where your money is stored. A reference variable doesn't contain an object; it contains a reference to an object. Furthermore, two or more reference variables can contain references to the same object but this is usually not a good idea.

Call the object's public accessor methods

Returning to the **Main** method in the class named **Props01** , [Listing 3](#) calls the object's public accessor methods named **setColor** and **getColor** twice in succession.

Note:

Listing 3 . Call the object's public accessor methods.

```
//Call public accessor methods
obj.setColor("Red");
Console.WriteLine("color: " +
obj.getColor());

//Call public accessor methods again with an
// invalid input value.
obj.setColor("Green");
Console.WriteLine("color: " +
obj.getColor());
```

The setColor method

The purpose of the method named **setColor** is to store a string value in the object. The calling code has no way of knowing how the value is stored because the implementation is hidden behind a **public** interface method. This is an example of encapsulation.

The string value to be stored is passed as a parameter to the method each time the method is called. An invalid string value was purposely passed as a parameter on the second call to the **setColor** method.

The **getColor** method

The purpose of the **getColor** method is to retrieve the string value previously stored in the object by the **setColor** method. Once again, the calling code has no way of knowing how the value is retrieved and returned because the implementation is hidden behind a **public** interface method.

The value that is retrieved by each call to the **getColor** method is displayed on the standard output device (the black screen).

Would be a property in Java

If this were a Java program, the combination of these two methods would constitute a property named **color** because the pair of methods matches the design pattern for properties in Java. However, that is not the case in C#. As you will see later, C# uses a different approach to identify properties. In C#, these are simply public accessor methods used for information hiding.

The definition of **setColor** and **getColor** in **TargetClass**

[Listing 4](#) defines the public **setColor** and **getColor** methods in the class named **TargetClass**.

Note:

Listing 4 . Definition of setColor and getColor methods in TargetClass.

```
private string colorData = "";

public void setColor(string data){
    //Validating code
    if(data.Equals("Red") ||
data.Equals("Blue")){
        //Accept incoming data value
        colorData = data;
    }else{
        //Reject incoming data value
        colorData = "Bad color";
    }//end if-else
}//end setColor
//-----
-----//

public string getColor(){
    return colorData;
}//end getColor
```

Typical method definitions

These two methods are typical of the method definition syntax in C#. The syntax is not too different from a function definition in C++ so you should have no trouble understanding the syntax.

Data validation

One of the reasons for hiding the implementation behind public accessor methods is to assure that only valid data is stored in the object. Therefore, public accessor methods that store data in the object often contain code that

validates the incoming data (as shown in [Listing 4](#)) before actually storing the data in the object.

The validation code in [Listing 4](#) will only accept incoming color data of "Red" or "Blue". If the incoming string doesn't match one of those two values, the string "Bad color" is stored in the object in place of the incoming value.

What do you do with invalid data

It usually isn't too difficult to write code to implement a set of rules to validate the incoming data. The hard part is figuring out what to do if the incoming data is not valid. The choices range all the way from flagging the data as invalid as shown in [Listing 4](#) to throwing an exception which, if not properly handled, will cause the program to terminate. The circumstances dictate the action to be taken.

Program output

The code in [Listing 3](#) calls the **setColor** method twice. The first call passes a valid string, "Red", as a parameter. The second call passes an invalid string, "Green", as a parameter. This causes the second and third lines of text in [Figure 1](#) to be displayed by the program.

Set, get, and display a property value with a valid value

As mentioned earlier, C# uses a special approach to identify properties. (You will see the code in **TargetClass** that accomplishes this shortly.) In the meantime, the code in [Listing 5](#) (which is still part of the **Main** method) first sets, then gets, and finally displays the value of a property named **height** belonging to the object referenced by the contents of the variable named **obj**.

Note:

Listing 5 . Set, get, and display a property value with a valid value.

```
obj.height = 20;  
Console.WriteLine("height: " + obj.height);
```

Looks can be deceiving

If you compare [Listing 5](#) with [Listing 1](#), you will see that there is essentially no difference in the syntax of the code in the two listings. The syntax in both listings suggests that a value is being directly assigned to a public instance variable belonging to the object. As we already know, that is true for [Listing 1](#). However, that is not true for [Listing 5](#).

A set Accessor method

Although it doesn't look like it, the code in [Listing 5](#) is actually calling a special [set Accessor](#) method that hides the implementation behind a public interface. As you will see shortly, this special method contains validation code similar what you saw in [Listing 4](#).

A get Accessor method

Once again, although it doesn't look like it, the last statement in [Listing 5](#) is actually calling a special [get Accessor](#) method belonging to the object to get the current value of the property. That method returns the value stored in the property. As before, the returned value is concatenated with a literal string and passed to the **WriteLine** method of the **Console** class for display on the black screen.

The set Accessor and get Accessor methods for the property named height

Returning to the class named **TargetClass** , the special *set Accessor* and *get Accessor* methods for the property named **height** are shown in [Listing 6](#).

Note:

Listing 6 . The set Accessor and get Accessor methods for the property named height.

```
private int heightData;

public int height{
    get{
        return heightData;
    }//end get

    set{
        //Validating code
        if(value < 84){
            heightData = value;
        }else{
            heightData = 0;
        }//end else
    }//end set

} //end height property
```

A hidden parameter named value

The value on the right side of the assignment operator in [Listing 5](#) arrives on the **set** side of the code in [Listing 6](#) as a hidden parameter named **value** .

Validating code

You can write whatever validating code is appropriate before assigning the incoming value to a private instance variable or perhaps storing it in a private data structure of some other type.

In this case, the value is accepted and stored in the private instance variable named **heightData** if it is less than 84 (the height in inches of a person that is seven feet tall). If the incoming value is greater than 83, it is not accepted and instead is flagged as invalid by storing 0 in the private instance variable.

A read-only property

You can omit the code on the **set** side if you need a read-only property.

Little more to say about this

There isn't much more that I can say to explain this syntax other than to tell you to memorize it. The code in [Listing 5](#) causes the fourth line of text shown in [Figure 1](#) to be displayed on the black screen.

Call manipulator method and display results

In addition to making it possible to set and get property values, objects often provide other public interface methods of varying complexity that make it possible to manipulate the data stored in the object in a variety of ways. In those cases, the using programmer needs access to good documentation that explains the behavior of such manipulator methods.

Returning to the **Main** method, [Listing 7](#) calls a manipulator method named **doubleHeight** (belonging to the object of type **TargetClass**) that is designed to double the value of the **height** property. Then [Listing 7](#) accesses and displays the new value of the **height** property.

Note:

Listing 7 . Call manipulator method and display results.

```
obj.doubleHeight();  
Console.WriteLine("double height: " +  
obj.height);
```

A manipulator method named **doubleHeight**

[Listing 8](#) shows the manipulator method named **doubleHeight** that is defined in the class named **TargetClass** .

Note:

Listing 8 . A simple manipulator method named **doubleHeight**.

```
public void doubleHeight(){  
    heightData *= 2;  
} //end doubleHeight  
//-----  
-----//  
  
} //end TargetClass
```

The method multiplies the current value in the private instance variable that is used to store the property named **height** by a factor of two, stores the modified value back into the same variable, and returns **void** .

The code in [Listing 7](#) causes the fifth line of text shown in [Figure 1](#) to be displayed.

[Listing 8](#) also signals the end of the class named **TargetClass** .

Set and get the property with an invalid value

Returning once more to the **Main** method, [Listing 9](#) attempts to set an invalid value into the property named **height**.

Note:

Listing 9 . Set and get the property named height with an invalid value.

```
obj.height = 100;  
Console.WriteLine("height: " + obj.height);
```

Then it retrieves and displays the value currently stored in that property.

As you saw in [Listing 6](#), when an attempt is made to set a value greater than 83 in the property, a value of 0 is set in the property to flag it as invalid.

The code in [Listing 9](#) causes the last line of text in [Figure 1](#) to be displayed on the black screen.

Pause until the user presses any key

The last statement in the **Main** method, shown in [Listing 10](#), causes the program to block and wait until the user presses a key. This causes the black screen to remain on the desktop until the user is finished viewing it.

Note:

Listing 10 . Pause until the user presses any key.

```
        Console.ReadKey();  
    }//end Main  
  
} //end class Props01
```

The end of the program

[Listing 10](#) signals the end of the **Main** method, the end of the class named **Props01** , and the end of the program. When the user presses a key, the black screen disappears from the desktop and the program terminates.

Run the program

I encourage you to copy the code from [Listing 11](#). Use that code to create a C# console application. Build and run the program. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Run my program

Click [here](#) to download a zip file containing my version of the program. Extract the folder named **Props01** from the zip file and save it somewhere on your disk. Start **Visual C# 2010 Express** and select **Open Project...** from the **File** menu. Navigate to the project folder and select the file with the extension of **.sln** . This should cause the project to open and be ready to run or debug as described in the earlier module titled [Getting Started](#).

Summary

You learned about abstraction, abstract data types, encapsulation, information hiding, class member access control, instance variables, local variables, class variables, reference variables, primitive variables, how C# identifies properties, and how to set and get properties in C#.

You also learned about public accessor methods, public manipulator methods, data validation by property setters and public accessor methods, and a few other things along the way.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Xna0106-Encapsulation in C#
- File: Xna0106.htm
- Published: 02/24/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation :: I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the C# program discussed in this module is provided in [Listing 11](#).

Note:

Listing 11 . The program named Props01.

```
/*Project Props01  
Copyright 2009, R.G.Baldwin
```

This program is designed to explore and explain
the use
of encapsulation and properties in C#.

Program output is:

text: Quit

color: Red

color: Bad color

height: 20

double height: 40

height: 0

```
*****  
*****/
```

```
using System;
```

```
namespace Props01{  
    class Props01{  
        static void Main(string[] args){  
            TargetClass obj = new TargetClass();  
  
            //Access a public instance variable  
            obj.text = "Quit";  
            Console.WriteLine("text: " + obj.text);
```

```

        //Call public accessor methods
        obj.setColor("Red");
        Console.WriteLine("color: " +
obj.getColor());

        //Call public accessor methods again with an
        // invalid input value.
        obj.setColor("Green");
        Console.WriteLine("color: " +
obj.getColor());

        //Set and get a property
        obj.height = 20;
        Console.WriteLine("height: " + obj.height);

        //Call manipulator method and display
results.
        obj.doubleHeight();
        Console.WriteLine("double height: " +
obj.height);

        //Set and get the property again with an
invalid
        // input value.
        obj.height = 100;
        Console.WriteLine("height: " + obj.height);

        //Pause until user presses any key.
        Console.ReadKey();
    }//end Main

} //end class Props01

//=====
=====//

```

```

public class TargetClass{
    //A public instance variable - not good
practice
    public string text;

    //This would be a property named color in
Java, but
    // not in C#
    private string colorData = "";
    public void setColor(string data){
        //Validating code
        if(data.Equals("Red") ||
data.Equals("Blue")){
            //Accept incoming data value
            colorData = data;
        }else{
            //Reject incoming data value
            colorData = "Bad color";
        }//end if-else
    }//end setColor
    //-----
-----//

    public string getColor(){
        return colorData;
    }//end getColor
    //-----
-----//

    //This is a C# property named height
    private int heightData;
    public int height{
        get{
            return heightData;
        }//end get
        set{
            //Validating code

```

```
        if(value < 84){
            heightData = value;
        }else{
            heightData = 0;
        }//end else
    }//end set
}//end height property
//-----
-----//

//This is a manipulator method
public void doubleHeight(){
    heightData *= 2;
}//end doubleHeight
//-----
-----//

}//end TargetClass
}//end namespace
```

-end-

Xna0108-Inheritance in C#

Learn how one class can extend another class and inherit all of the properties, events, and methods defined in that class and all of its superclasses; that even though a class may be extended into another class, it remains viable and can be instantiated in its own right; that inheritance is hierarchical with the overall hierarchy being rooted in a class named Object; that C# does not support multiple inheritance; about the ISA and HASA relationships.

Revised: Thu May 05 14:27:52 CDT 2016

This page is part of a Book titled [XNA Game Studio](#).

Table of Contents

- [Table of Contents](#)
- [Preface](#)
 - [The three main characteristics of an object-oriented program](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [A new class can extend an existing class](#)
 - [What is inherited?](#)
 - [The superclass remains viable](#)
 - [A hierarchy of classes](#)
 - [Members of the System.Object class](#)
 - [The Button class](#)
 - [The Object class is the default superclass](#)
 - [An orderly hierarchy.](#)

- [The airship hierarchy](#)
 - [The Balloon class](#)
 - [The Airplane class](#)
 - [Three types of objects](#)
 - [From the general to the specialized](#)
- [Single and multiple inheritance](#)
- [The ISA relationship](#)
- [The HASA relationship](#)
- [Preview](#)
 - [The airship hierarchy](#)
- [Discussion and sample code](#)
 - [Will explain in fragments](#)
 - [Four class definitions](#)
 - [The Airship class](#)
 - [The Balloon class](#)
 - [The new material](#)
 - [The effect of extending a class](#)
 - [Airship extends Object](#)
 - [The beginning of the Driver class](#)
 - [Code common to all four classes](#)
 - [Instantiate a new Balloon object](#)
 - [Set properties in the Balloon object](#)
 - [Get and display property values](#)
 - [The program output](#)
 - [The remaining code](#)
 - [No sharing of properties](#)
- [Run the program](#)

- [Run my program](#)
- [Summary](#)
- [Miscellaneous](#)
- [Complete program listing](#)

Preface

This module is one in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

An earlier module titled [Getting Started](#) provided information on how to get started programming with Microsoft's XNA Game Studio.

The three main characteristics of an object-oriented program

Object-oriented programs exhibit three main characteristics:

- Encapsulation
- Inheritance
- Polymorphism

I will explain and illustrate inheritance along with some related topics in this module.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Output from the program named Airship01.

Listings

- [Listing 1](#). The Airship class.
- [Listing 2](#). The Balloon class.
- [Listing 3](#). The beginning of the Driver class.
- [Listing 4](#). The program named Airship01.

General background information

The first of the three major characteristics of an object-oriented program is *encapsulation*. I explained *encapsulation* in an earlier module. The second of the three is *inheritance*, followed by *polymorphism*. I will explain *inheritance* in this module and will explain *polymorphism* in a future module.

A new class can extend an existing class

A class can be defined to inherit the properties, events, and methods of another class. From a syntax viewpoint, this is accomplished using the `:` (colon) operator (see [Listing 2](#)).

The class being extended or inherited from is often called the *base class* or the *superclass*. The new class is often called the *derived class* or the *subclass*.

What is inherited?

The subclass inherits the data representation and behavior of the superclass (*and all of its superclasses*). However, the subclass can modify the behavior of inherited methods by overriding them, provided that they were declared *virtual* by the original author. (*That will be one of the topics in a*

future module on polymorphism.) The subclass can also add new data representation and behavior that is unique to its own purposes.

The superclass remains viable

A program can instantiate objects of a superclass as well as instantiating objects of its subclasses. From a practical viewpoint, the superclass doesn't even know that it has been extended.

A hierarchy of classes

Inheritance is hierarchical. By that, I mean that a class may be the subclass of one (*and only one*) other class and may be the superclass of one or more other classes.

The overall inheritance hierarchy has a single root in the [System.Object](#) class. In other words, the [System.Object](#) class is the common ancestor for every other class.

Members of the System.Object class

The **System.object** class defines the following [eight methods](#), which are inherited by every other class:

- **Equals** - Overloaded. Determines whether two Object instances are equal.
 - **Equals(Object)** - Determines whether the specified Object is equal to the current Object.
 - **Equals(Object, Object)** - Determines whether the specified Object instances are considered equal.
- **Finalize** - Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage

collection.

- **GetHashCode** - Serves as a hash function for a particular type.
- **GetType** - Gets the Type of the current instance.
- **MemberwiseClone** - Creates a shallow copy of the current Object.
- **ReferenceEquals** - Determines whether the specified Object instances are the same instance.
- **ToString** - Returns a String that represents the current Object.

Because these eight methods are inherited by every other class, they are always available for you to use in your code. (Possibly the most frequently used of these methods is the **ToString** method.)

The **Button** class

Moving down a single path in the inheritance hierarchy, we find that the family tree for the **Button** class in the **System.Windows.Forms** namespace is as follows:

- System. **Object**
- System.MarshalByRefObject
- System.ComponentModel.Component
- System.Windows.Forms.Control
- System.Windows.Forms.ButtonBase
- System.Windows.Forms. **Button**

If you were to examine the documentation for each of the classes in the **Button** class' family tree, you would probably find that each class is more specialized than its superclass. For example, the **Object** class is very generic and the **Button** class is very specialized. Generally speaking, classes become more specialized as you move down the hierarchy beginning with the **Object** class.

The **Object** class is the default superclass

When you define a new class, it becomes an immediate subclass of the **Object** class by default unless you cause your new class to extend some other class.

An orderly hierarchy

The C# inheritance mechanism allows you build an orderly hierarchy of classes to supplement the classes that are already in the class library.

When several of your abstract data types have characteristics in common, you can design their commonalities into a single superclass and separate their unique characteristics into unique subclasses. This is one of the purposes of inheritance.

The airship hierarchy

For example, suppose you are building a program dealing with airships. All airships have altitude and range properties in common. Therefore, you could build a base **Airship** class containing data and methods having to do with range and altitude.

From this superclass, you could derive a **Balloon** class and an **Airplane** class.

The Balloon class

The **Balloon** class might add properties and methods dealing with passenger capacity and what makes it go up (*helium, hydrogen, or hot air*) . Objects of the **Balloon** class would then be able to deal with altitude, range, passenger capacity, and what makes it go up.

The Airplane class

The **Airplane** class might add properties and methods dealing with engine type (*jet or propeller*) and cargo capacity. Objects of the **Airplane** class could then deal with altitude, range, engine type, and cargo capacity.

Three types of objects

Having created this hierarchy of classes, you could instantiate objects of type **Airship** , **Balloon** , and **Airplane** with the objects of each type having properties and methods to deal with those special characteristics of the flying machine indicated by the name of the class.

From the general to the specialized

You may have noticed that in this hierarchical class structure, inheritance causes the structure to grow in a direction from most general to more specialized. This is typical.

Single and multiple inheritance

C++ and some other object-oriented programming languages allow for multiple inheritance. This means that a new class can extend more than one superclass. This has advantages in some cases, but can lead to difficulties in other cases.

C# does not support multiple inheritance. Instead it supports a different mechanism called an *interface* that provides most of the benefits of multiple inheritance without most of the problems. I will explain the C# interface in a future module.

The ISA relationship

You will sometimes hear people speak of the **ISA** relationship when discussing OOP (*such as in he **is a** hero*) . The source of this terminology is more fundamental than you may at first suspect.

Object-oriented designers often strive to use inheritance to model relationships where a subclass "*is a kind of*" the superclass. For example, a car "*is a kind of*" vehicle. A programmer "*is a kind of*" employee which in turn "*is a kind of*" person. An airplane "*is a kind of*" airship and so is a hot-air balloon.

This relationship is called the **ISA** relationship. It's that simple.

The HASA relationship

If you were to define a class that more fully represents an airplane, you might choose to break certain parts of the airplane out into separate objects and to incorporate them by reference into your **Airplane** class.

For example, you might incorporate a **Cockpit** object, a **LandingGear** object, a **Propeller** object, etc. Then, (even though it wouldn't be good grammar), you could say that an object of your **Airplane** class "*has a*" **Cockpit** object, "*has a*" **LandingGear** object, etc. This is often referred to as a HASA relationship. For example, an airplane ISA airship and HASA cockpit.

Preview

The airship hierarchy

A little earlier I explained an airship hierarchy involving an **Airship** class, a **Balloon** class, and an **Airplane** class. I will present and explain a sample program that implements that hierarchy. The output from the program is shown in [Figure 1](#). I will refer back to [Figure 1](#) in the paragraphs that follow.

Note:

Figure 1 . Output from the program named Airship01.

Balloon

range = 5 miles

altitude = 500 feet

passenger capacity = 5

lift media = Hot Air

Airplane

range = 5000 miles

altitude = 35000 feet

cargo capacity = 20000 pounds

engine type = jet

Discussion and sample code

Will explain in fragments

I will explain this program in fragments. A complete listing of the program is provided in [Listing 4](#) near the end of the module.

Four class definitions

This program consists of four class definitions:

- Airship
- Balloon
- Airplane
- Driver

The **Airship** class defines properties that are common to machines that fly:

- Range
- Altitude

The **Balloon** class extends the **Airship** class and defines properties that are peculiar to airships that are lighter than air:

- Passenger capacity
- Lift Media (hot air, helium, or hydrogen)

The **Airplane** class extends the **Airship** class and defines properties that are peculiar to airplanes:

- Cargo capacity
- Engine type

The **Driver** class instantiates objects of the **Balloon** class and the **Airplane** class and exercises their **set** and **get** methods.

The **Airship** class

[Listing 1](#) shows the **Airship** class in its entirety.

Note:

Listing 1 . The Airship class.

```
//Define common properties in the base class.
class Airship {
    private int rangeData = 0;
    private int altitudeData = 0;

    public int range {
        get {
            return rangeData;
        } //end get
        set {
```



```

        rangeData = value;
    }//end set
}//end range property

public int altitude {
    get {
        return altitudeData;
    }//end get
    set {
        altitudeData = value;
    }//end set
}//end altitude property
}//end class Airship

```

There is nothing in [Listing 1](#) that you haven't seen in earlier modules. This class provides **set** and **get** methods for two properties named **range** and **altitude** .

The **Balloon** class

[Listing 2](#) shows the **Balloon** class in its entirety.

Note:

Listing 2 . The Balloon class.

```

//Define unique properties in the subclass.
class Balloon : Airship {
    private int passengerCapacityData;
    private String liftMediaData;

    public int passengerCapacity {

```

```

        get {
            return passengerCapacityData;
        } //end get
        set {
            passengerCapacityData = value;
        } //end set
    } //end passengerCapacity property

    public String liftMedia {
        get {
            return liftMediaData;
        } //end get
        set {
            liftMediaData = value;
        } //end set
    } //end liftMedia property
} //end Balloon class

```

The new material

The only thing in [Listing 2](#) that is new to this module is the colon that appears between the words *Balloon* and *Airship* on the second line. This is the C# way of specifying that the class named **Balloon** extends or inherits from the class named **Airship**. In this case, the **Balloon** class is the subclass or derived class and the **Airship** class is the superclass or base class, depending on which flavor of jargon you prefer.

The effect of extending a class

The effect of having the **Balloon** class extend the **Airship** class is different from anything that you have seen in previous modules. When one class extends another class, the new class inherits all of the properties, events, and methods of the superclass and all of its superclasses.

Airship extends Object

In this case, the **Airship** class extends the **Object** class by default. Therefore, an object instantiated from the **Balloon** class contains:

- The two properties defined in the **Balloon** class in [Listing 2](#).
- The two properties defined in the **Airship** class in [Listing 1](#).
- The eight methods defined in the **Object** class discussed [earlier](#).

The beginning of the Driver class

At this point, I am going to show and explain the first half of the **Driver** class (see [Listing 3](#)) and relate it to the program output shown in [Figure 1](#).

Note:

Listing 3 . The beginning of the Driver class.

```
using System; namespace Airship01 {  
  
    //Define a class to exercise the Balloon class  
    and the  
    // Airplane class.  
    class Driver {  
        static void Main(string[] args) {  
  
            Balloon balloon = new Balloon();  
            balloon.range = 5;  
            balloon.altitude = 500;  
            balloon.passengerCapacity = 5;  
            balloon.liftMedia = "Hot Air";  
  
            Console.WriteLine("Balloon");  
            Console.WriteLine(  
                "range = " + balloon.range + "  
miles");
```

```
        Console.WriteLine(
            "altitude = " + balloon.altitude + "
feet");
        Console.WriteLine("passenger capacity = "
            +
            balloon.passengerCapacity);
        Console.WriteLine("lift media = "
            +
            balloon.liftMedia);
```

For convenience and because of their small sizes, I elected to define all four classes in the same file as the file that contains the **Driver** class with the **Main** method. That is not a requirement, however, and on large projects you may want to put each class definition in its own file.

Code common to all four classes

The single line of code at the very beginning of [Listing 3](#) applies to all four classes. You have seen this before so it should not be new to you.

Instantiate a new Balloon object

The **Main** method begins by instantiating a new object of the **Balloon** class and saving its reference in a reference variable of type **Balloon** named **balloon** . This reference will be used later to access the new object.

Set properties in the Balloon object

Then the **Main** method calls the four **set** methods belonging to the new **Balloon** object, using them to set values into the four properties belonging to the **Balloon** object.

Note: Remember that the **set** methods that hide two of the properties are defined in the **Airship** class and are inherited into the **Balloon** class. The other two **set** methods are defined in the **Balloon** class.

Get and display property values

After that, the **Main** method calls the four **get** methods belonging to the new **Balloon** object, using them to get and display values from the four properties belonging to the new **Balloon** object.

Note: The **get** methods that hide two of the properties are defined in the **Airship** class and are inherited into the **Balloon** class. The other two **get** methods are defined in the **Balloon** class.

The program output

The code in [Listing 3](#) produces the first five lines of output text shown in [Figure 1](#).

The remaining code

You can view the remaining code in the **Driver** class and the code in the **Airplane** class in [Listing 4](#). If you understand my explanation of [Listing 1](#), [Listing 2](#), and [Listing 3](#) above, you should have no difficulty understanding the behavior of the code in the **Airplane** class and the remaining code in the **Driver** class.

No sharing of properties

There is an important point to remember, however. Even though the **Balloon** and **Airplane** classes each inherit the **range** and **altitude** properties from the **Airship** class, objects instantiated from the **Balloon** and **Airplane** classes do not physically share these two properties. Instead, each object has its own copy of the **range** property and its own copy of the **altitude** property. The only thing shared by the two objects is part of the blueprint from which each object is constructed.

Basically there is no sharing of anything among objects until the **static** keyword shows up and at that point, some sharing does take place. The use of the **static** keyword is a topic for a future module.

Run the program

I encourage you to copy the code from [Listing 4](#). Use that code to create a C# console project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Run my program

Click [here](#) to download a zip file containing my version of the program. Extract the folder named **Airship01** from the zip file and save it somewhere on your disk. Start **Visual C# 2010 Express** and select **Open Project...** from the **File** menu. Navigate to the project folder and select the file with the extension of **.sln**. This should cause the project to open and be ready to run or debug as described in the earlier module titled [Getting Started](#).

Summary

In this module, you learned how one class can extend another class and inherit all of the properties, events, and methods defined in that class and all of its superclasses. You learned that even though a class may be extended into another class, it remains viable and can be instantiated in its own right. You learned that inheritance is hierarchical with the overall hierarchy being

rooted in a class named **Object** . You learned that C# does not support multiple inheritance. You learned about the ISA and HASA relationships.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Xna0108-Inheritance in C#
- File: Xna0108.htm
- Published: 02/27/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the C# program discussed in this module is provided in [Listing 4](#).

Note:

Listing 4 . The program named Airship01.

```
/*Project Airship01
 * Illustrates inheritance
 *
 *****/

using System;

namespace Airship01 {

    //Define a class to exercise the Balloon class
    and the
    // Airplane class.
    class Driver {
        static void Main(string[] args) {
            Balloon balloon = new Balloon();
            balloon.range = 5;
            balloon.altitude = 500;
            balloon.passengerCapacity = 5;
            balloon.liftMedia = "Hot Air";

            Console.WriteLine("Balloon");
            Console.WriteLine(
                "range = " + balloon.range + "
miles");
            Console.WriteLine(
                "altitude = " + balloon.altitude + "
```



```

feet");
    Console.WriteLine("passenger capacity = "
                      +
balloon.passengerCapacity);
    Console.WriteLine("lift media = "
                      +
balloon.liftMedia);

    Airplane airplane = new Airplane();
    airplane.range = 5000;
    airplane.altitude = 35000;
    airplane.cargoCapacity = 20000;
    airplane.engineType = "jet";

    Console.WriteLine(""); //blank line
    Console.WriteLine("Airplane");
    Console.WriteLine(
        "range = " + airplane.range + "
miles");
    Console.WriteLine(
        "altitude = " + airplane.altitude + "
feet");
    Console.WriteLine("cargo capacity = "
                      + airplane.cargoCapacity + "
pounds");
    Console.WriteLine("engine type = "
                      +
airplane.engineType);

    //Pause and wait for the user to press any
key.
    Console.ReadKey();
} //end Main
} //end class Driver

//=====
=====//

```

```

//Define common properties in the base class.
class Airship {
    private int rangeData = 0;
    private int altitudeData = 0;

    public int range {
        get {
            return rangeData;
        } //end get
        set {
            rangeData = value;
        } //end set
    } //end range property

    public int altitude {
        get {
            return altitudeData;
        } //end get
        set {
            altitudeData = value;
        } //end set
    } //end altitude property

} //end class Airship

//=====
====//

//Define unique properties in the subclass.
class Balloon : Airship {
    private int passengerCapacityData;
    private String liftMediaData;

    public int passengerCapacity {
        get {
            return passengerCapacityData;

```

```

    }//end get
    set {
        passengerCapacityData = value;
    }//end set
} //end passengerCapacity property

public String liftMedia {
    get {
        return liftMediaData;
    }//end get
    set {
        liftMediaData = value;
    }//end set
} //end liftMedia property
} //end Balloon class

```

```

//=====
====//

```

```

//Define unique properties in the subclass.
class Airplane : Airship {
    private int cargoCapacityData;
    private String engineTypeData;

    public int cargoCapacity {
        get {
            return cargoCapacityData;
        }//end get
        set {
            cargoCapacityData = value;
        }//end set
    } //end cargoCapacity property

    public String engineType {
        get {
            return engineTypeData;
        }//end get
    }
}

```

```
        set {
            engineTypeData = value;
        }//end set
    }//end engineType property
} //end Airplane class

//=====
====//
} //end namespace Airship01
```

-end-

Xna0110-Polymorphism Based on Overloaded Methods

Learn that overloaded methods have the same name and different formal argument lists; that the word polymorphism means something like one name, many forms; that polymorphism manifests itself in C# in the form of multiple methods having the same name; that polymorphism manifests itself in three distinct forms in C#: method overloading, method overriding through class inheritance, and method overriding through interface inheritance.

Revised: Thu May 05 15:30:44 CDT 2016

This page is part of a Book titled [XNA Game Studio](#).

Table of Contents

- [Table of Contents](#)
- [Preface](#)
 - [The three main characteristics of an object-oriented program](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [What is polymorphism?](#)
 - [How does C# implement polymorphism?](#)
 - [The class named Object](#)
 - [Multiple methods with the same name](#)
 - [Compile-time and runtime polymorphism, what's the difference?](#)
 - [Every class extends some other class](#)
 - [A class hierarchy](#)
 - [Methods in the Object class](#)
 - [Some methods are overloaded](#)
 - [Some methods are meant to be overridden](#)
 - [Some methods are meant to be used as is](#)

- [Three distinct forms of polymorphism](#)
- [Method overloading](#)
 - [Duplicate method names](#)
 - [Compile-time polymorphism](#)
 - [Selection based on the argument list](#)
- [Preview](#)
 - [Keeping it short and simple](#)
 - [A sample program](#)
 - [Within the class and the hierarchy](#)
 - [Class B extends class A, which extends Object](#)
 - [Designed to illustrate method overloading](#)
- [Discussion and sample code](#)
 - [Will discuss in fragments](#)
 - [The class named A](#)
 - [Redundant code](#)
 - [The method named m\(\)](#)
 - [The class named B](#)
 - [Overloaded methods](#)
 - [The driver class](#)
 - [Call all three overloaded methods](#)
 - [One version is inherited](#)
 - [Two versions defined in class B](#)
 - [The output](#)
 - [Parameters are not displayed](#)
- [Run the program](#)

- [Run my program](#)
- [Summary](#)
- [Miscellaneous](#)
- [Complete program listing](#)

Preface

This module is one in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

An earlier module titled [Getting Started](#) provided information on how to get started programming with Microsoft's XNA Game Studio.

The three main characteristics of an object-oriented program

Object-oriented programs exhibit three main characteristics:

- Encapsulation
- Inheritance
- Polymorphism

I have explained encapsulation and inheritance in previous modules. I will begin the explanation of polymorphism in this module. However, polymorphism is a complex topic and several modules will be required to complete the explanation.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Screen output from the project named Polymorph01.

Listings

- [Listing 1](#). Class A from the project named Polymorph01.
- [Listing 2](#). Class B from the project named Polymorph01.
- [Listing 3](#). Class Polymorph01 from the project named Polymorph01.
- [Listing 4](#). Project Polymorph01.

General background information

What is polymorphism?

The meaning of the word polymorphism is something like *one name, many forms* .

How does C# implement polymorphism?

Polymorphism manifests itself in C# in the form of multiple methods having the same name.

In some cases, multiple methods have the same name, but have different formal argument lists. These are *overloaded* methods.

In other cases, multiple methods have the same name, same return type, and same formal argument list. These are *overridden* methods.

This module concentrates on the use of method *overloading* to achieve *compile-time* polymorphism.

The class named Object

As you learned in an earlier module, every class in C# is a direct or indirect subclass of the class named **Object** . Methods defined in the **Object** class are inherited into all other classes. Some of those inherited methods may be *overridden* to make their behavior more appropriate for objects instantiated from the new subclasses. However, this module is not about *overriding* methods. Instead it is about *overloading* methods. I will cover method overriding in future modules.

Multiple methods with the same name

Overloaded methods have the same name and different formal argument lists. They may or may not have the same return type.

Polymorphism manifests itself in C# in the form of multiple methods having the same name. As mentioned above, this module concentrates on method *overloading* , sometimes referred to as *compile-time polymorphism* . Subsequent modules concentrate on method *overriding* , sometimes referred to as *runtime polymorphism* .

Compile-time and runtime polymorphism, what's the difference?

During the compilation and execution of polymorphic code, the compiler and the runtime system must decide which of two or more methods having the same name in the same scope must be executed. With method overloading, that decision is made when the program is compiled. With method overriding, that decision is deferred and made at runtime. Hence we have the terms *compile-time polymorphism* and *runtime polymorphism* .

Every class extends some other class

Every class in C# (except for the class named **Object**) extends some other class. If you don't explicitly specify the class that your new class extends, it will automatically extend the class named **Object** .

A class hierarchy

Thus, all classes in C# exist in a class hierarchy where the class named **Object** forms the root of the hierarchy.

Some classes extend **Object** directly, while other classes are subclasses of **Object** further down the hierarchy (they extend classes that extend classes that extend **Object**).

Methods in the Object class

You learned in an earlier module that the class named **Object** defines default versions of the following methods and that every class in the hierarchy inherits them:

- **Equals** - Overloaded. Determines whether two Object instances are equal.
 - **Equals(Object)** - Determines whether the specified Object is equal to the current Object.
 - **Equals(Object, Object)** - Determines whether the specified Object instances are considered equal.
- **Finalize** - Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
- **GetHashCode** - Serves as a hash function for a particular type.
- **GetType** - Gets the Type of the current instance.
- **MemberwiseClone** - Creates a shallow copy of the current Object.
- **ReferenceEquals** - Determines whether the specified Object instances are the same instance.
- **ToString** - Returns a String that represents the current Object.

Some methods are overloaded

The two methods named **Equals** are defined as overloaded methods in the **Object** class. (They have the same name and different formal argument lists.)

Some methods are meant to be overridden

Some of these methods are intended to be overridden for various purposes. This includes **GetHashCode** , and **ToString** .

Some methods are meant to be used as is

However, some of the methods, such as **GetType** are intended to be used as is without overriding.

Three distinct forms of polymorphism

From a practical programming viewpoint, polymorphism manifests itself in three distinct forms in C#:

- Method overloading
- Method overriding through class inheritance
- Method overriding through interface inheritance

Method overloading

I will begin the discussion of polymorphism with method overloading, which is the simplest of the three. I will cover method overloading in this module and will cover polymorphism based on overridden methods using class inheritance and interface inheritance in future modules.

Duplicate method names

C# allows you to have two or more method definitions in the same scope with the same name, provided that they have different formal argument lists. This is method *overloading* .

Compile-time polymorphism

Some authors refer to method overloading as a form of *compile-time polymorphism* , as distinguished from *run-time polymorphism* . This distinction comes from the fact that, for each method call, the compiler determines which method (from a group of overloaded methods) will be executed. This decision is made when the program is compiled.

In contrast, with method overriding, the determination of which overridden method to execute isn't made until runtime.

Selection based on the argument list

In practice, with overloaded methods, the compiler simply examines the types, number, and order of the parameters being passed in a method call and selects the overloaded method having a matching formal argument list.

Preview

Keeping it short and simple

In these modules on polymorphism, I will explain sample programs that are as short and as simple as I know how to make them, while still illustrating the important points regarding polymorphism. My objective is to make the polymorphic concepts as clear as possible without having those concepts clouded by other programming issues.

Because of their simplicity, these programs aren't likely to resemble problems that you will encounter in the real world. I will simply ask you to

trust me when I tell you that polymorphism has enormous application in real world programming.

A sample program

I will explain a sample program named **Polymorph01** to illustrate method overloading. A complete listing of the program is provided in [Listing 4](#) near the end of the module.

Within the class and the hierarchy

Method overloading can occur both within a class definition, and vertically within the class inheritance hierarchy.

In other words, an overloaded method can be inherited into a class that defines other overloaded versions of the method.

The program named **Polymorph01** illustrates both aspects of method overloading.

Class B extends class A, which extends Object

Upon examination of the program, you will see that the class named **A** extends the class named **Object** . You will also see that the class named **B** extends the class named **A** .

The class named **Polymorph01** is a driver class whose **Main** method exercises the methods defined in the classes named **A** and **B** .

Designed to illustrate method overloading

Once again, this program is not intended to correspond to any particular real-world scenario. Rather, it is a very simple program designed specifically to illustrate method overloading.

Discussion and sample code

Will discuss in fragments

As usual, I will discuss this program in fragments. A complete listing is provided in [Listing 4](#) near the end of the module.

The class named A

The code in [Listing 1](#) defines the class named **A** , which explicitly extends **Object** .

Note:

Listing 1 . Class A from the project named Polymorph01.

```
using System;

class A : Object {
    public void m() {
        Console.WriteLine("m()");
    } //end method m()
} //end class A
```

Redundant code

Recall that explicitly extending **Object** is not required (but it also doesn't hurt anything).

The class named **A** would extend the class named **Object** by default unless the class named **A** explicitly extends some other class.

The method named **m()**

The code in [Listing 1](#) defines a method named **m()** . Note that this version of the method has an empty argument list (it doesn't receive any parameters when it is executed). The behavior of the method is simply to display a message indicating that it is executing.

The class named **B**

[Listing 2](#) contains the definition for the class named **B** . This class extends the class named **A** and inherits the method named **m** defined in the class named **A** .

Note:

Listing 2 . Class B from the project named Polymorph01.

```
class B : A {  
    public void m(int x) {  
        Console.WriteLine("m(int x)");  
    }//end method m(int x)  
    //-----//  
  
    public void m(String y) {  
        Console.WriteLine("m(String y)");  
    }//end method m(String y)  
}//end class B
```

Overloaded methods

In addition to the inherited method named **m** , the class named **B** defines two additional overloaded versions of the method named **m** :

- m(int x)
- m(String y)

Note that each of these versions of the method receives a single parameter and the type of the parameter is different in each case.

As with the version of the method having the same name defined in the class named **A** , the behavior of each of these two methods is simply to display a message indicating that it is executing.

The driver class

[Listing 3](#) contains the definition of the driver class named **Polymorph01** .

Note:

Listing 3 . Class Polymorph01 from the project named Polymorph01.

```
public class Polymorph01 {  
    public static void Main() {  
        B var = new B();  
        var.m();  
        var.m(3);  
        var.m("String");  
  
        //Pause until the user presses a key.  
        Console.ReadKey();  
    }//end Main  
}  
}  
}
```


Call all three overloaded methods

The code in the **Main** method

- Instantiates a new object of the class named **B**
- Successively calls each of the three overloaded versions of the method named **m** on the reference to that object.

One version is inherited

The overloaded version of the method named **m** , defined in the class named **A** , is inherited into the class named **B** . Hence, it can be called on a reference to an object instantiated from the class named **B** .

Two versions defined in class B

The other two versions of the method named **m** are defined in the class named **B** . Thus, they also can be called on a reference to an object instantiated from the class named **B** .

The output

As you would expect from the code that you examined for each of the three methods, the output produced by sending messages to the object asking it to execute each of the three overloaded versions of the methods named **m** is shown in [Figure 1](#).

Note:

Figure 1 . Screen output from the project named Polymorph01.

```
m()  
m(int x)
```

```
m(String y)
```

Parameters are not displayed

The values of the parameters passed to the methods do not appear in the output. In this program, the parameters are used solely to make it possible for the compiler to select the correct version of the overloaded method to execute in each case.

Note: In a real program, however, the parameters would normally be used by the code in the method for some useful purpose.

This output confirms that each overloaded version of the method was properly selected for execution based on the matching of method parameters to the formal argument list of each method.

Run the program

I encourage you to copy the code from [Listing 4](#). Use that code to create a C# console project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Run my program

Click [here](#) to download a zip file containing my version of the program. Extract the folder named **Polymorph01** from the zip file and save it somewhere on your disk. Start **Visual C# 2010 Express** and select **Open Project...** from the **File** menu. Navigate to the project folder and select the file with the extension of **.sln** . This should cause the project to open and be

ready to run or debug as described in the earlier module titled [Getting Started](#).

Summary

Overloaded methods have the same name and different formal argument lists. They may or may not have the same return type.

The word polymorphism means something like *one name, many forms*. Polymorphism manifests itself in C# in the form of multiple methods having the same name.

Polymorphism manifests itself in three distinct forms in C#:

- Method overloading
- Method overriding through class inheritance
- Method overriding through interface inheritance

This module concentrates on method overloading, sometimes referred to as *compile-time polymorphism*.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Xna0110-Polymorphism Based on Overloaded Methods
- File: Xna0110.htm
- Published: 02/27/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the C# program discussed in this module is provided in [Listing 4](#).

Note:

Listing 4 . Project Polymorph01.

```
/*Project Polymorph01  
Copyright 2009, R.G.Baldwin
```

This program illustrates method overloading, both within a class, and up the inheritance hierarchy.

Program output is:

```

m()
m(int x)
m(String y)
*****/
using System;

class A : Object {
    public void m() {
        Console.WriteLine("m()");
    } //end method m()
} //end class A
//=====//

class B : A {
    public void m(int x) {
        Console.WriteLine("m(int x)");
    } //end method m(int x)
    //-----//

    public void m(String y) {
        Console.WriteLine("m(String y)");
    } //end method m(String y)
} //end class B
//=====//

public class Polymorph01 {
    public static void Main() {
        B var = new B();
        var.m();
        var.m(3);
        var.m("String");

        //Pause until the user presses
        // a key.
        Console.ReadKey();
    } //end Main

```

```
//end class Polymorph01
```

-end-

Xna0112-Type Conversion, Casting, and Assignment Compatibility
Learn about assignment compatibility, type conversion, and casting for both primitive and reference types. Also learn about the relationships among reference types, method invocations, and the location in the class hierarchy where a method is defined.

Revised: Fri May 06 14:33:08 CDT 2016

This page is part of a Book titled [XNA Game Studio](#).

Table of Contents

- [Table of Contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Listings](#)
- [General background information](#)
 - [Type conversion](#)
 - [Assignment compatibility](#)
 - [Successful cast depends on class hierarchy](#)
 - [The generic type Object](#)
 - [Calling a method on an object](#)
 - [Assignment compatibility and type conversion](#)
 - [Type conversion and the cast operator](#)
 - [Applying a cast operator](#)
 - [Primitive values and type conversion](#)
 - [Boolean types](#)
 - [Numeric primitive types](#)
 - [Assignment compatibility for references](#)

- [Type Object is completely generic](#)
 - [Converting reference types with a cast](#)
 - [Downcasting](#)
- [Preview](#)
- [Discussion and sample code](#)
 - [The class named A](#)
 - [The class named B](#)
 - [The method named m](#)
 - [The class named C](#)
 - [The driver class](#)
 - [An object of the class named B](#)
 - [Automatic type conversion](#)
 - [Only part of the story](#)
 - [An illegal operation](#)
 - [A compiler error](#)
 - [An important rule](#)
 - [This case violates the rule](#)
 - [The solution is a downcast](#)
 - [Still doesn't solve the problem](#)
 - [What is the problem here?](#)
 - [The real solution](#)
 - [A few odds and ends](#)
 - [A legal operation](#)
 - [Cannot be assigned to type C](#)
 - [A runtime exception](#)
 - [Another failed attempt](#)
 - [The end of the program](#)

- [Run the program](#)
- [Run my program](#)
- [Summary](#)
- [Miscellaneous](#)
- [Complete program listing](#)

Preface

This module is one in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

An earlier module titled [Getting Started](#) provided information on how to get started programming with Microsoft's XNA Game Studio.

Object-oriented programs exhibit three main characteristics:

- Encapsulation
- Inheritance
- Polymorphism

I have explained encapsulation, inheritance, and compile-time polymorphism in earlier modules. Before I can explain runtime polymorphism, however, I need to step back and explain some concepts involving type conversion, casting, and assignment compatibility.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Listings while you are reading about them.

Listings

- [Listing 1](#). Source code for class A.
- [Listing 2](#). Source code for class B.
- [Listing 3](#). Source code for class C.
- [Listing 4](#). Beginning of the class named Polymorph02.
- [Listing 5](#). Try to call method m on variable var.
- [Listing 6](#). Try a downcast to type A.
- [Listing 7](#). Try a downcast to type B.
- [Listing 8](#). Assign var to v1.
- [Listing 9](#). Cannot be assigned to C.
- [Listing 10](#). Another failed attempt.
- [Listing 11](#). Project Polymorph02.

General background information

Type conversion

This module explains type conversion for both primitive and reference types.

Assignment compatibility

A value of a particular type may be assignment-compatible with a variable of another type. If so, the value can be assigned directly to the variable.

If not, it may be possible to perform a cast on the value to change its type and assign it to the variable as the new type.

Successful cast depends on class hierarchy

With regard to reference types, whether or not a cast can be successfully performed depends on the relationship of the classes involved in the class hierarchy.

The generic type **Object**

A reference to any object can be assigned to a reference variable of the type **Object** , because the **Object** class is a superclass of every other class.

Note: In other words, an object instantiated from any class is assignment-compatible with the type **Object** .

Calling a method on an object

Whether or not a method can be called on a reference to an object depends on the *current type* of the reference and the location in the class hierarchy where the method is defined.

In order to use a reference of a class type to call a method, the method must be defined at or above that class in the class hierarchy.

Assignment compatibility and type conversion

As background, for understanding runtime polymorphism, you need to understand *assignment compatibility* and *type conversion* .

As I mentioned earlier, a value of a given type is assignment-compatible with another type if a value of the first type can be successfully assigned to a variable of the second type.

Type conversion and the cast operator

In some cases, type conversion happens automatically. In other cases, type conversion must be forced through the use of a cast operator.

A cast operator is a unary operator, which has a single right operand. The physical representation of the cast operator is the name of a type enclosed by a pair of matching parentheses, as in:

`(int)`

Applying a cast operator

Applying a cast operator to the name of a variable doesn't actually change the type of the variable. However, it does cause the contents of the variable to be treated as a different type for the evaluation of the expression in which the cast operator is contained.

Primitive values and type conversion

Assignment compatibility issues come into play for both primitive types and reference types.

Boolean types

To begin with, values of type **bool** can only be assigned to variables of type **bool** (you cannot change the type of a **bool**). Thus, a value of type **bool** is not assignment-compatible with a variable of any other type.

Numeric primitive types

In general, numeric primitive values can be assigned to (are assignment-compatible with) a variable of a type whose numeric range is as wide as or wider than the range of the type of the value. In that case, the type of the value is automatically converted to the type of the variable.

Note: For example, types **sbyte** and **short** can be assigned to a variable of type **int** , because type **int** has a wider range than either type **sbyte** or type **short** .

On the other hand, a primitive numeric value of a given type **cannot** be assigned to (is not assignment-compatible with) a variable of a type with a narrower range than the type of the value.

However, it is possible to use a cast operator to force a type conversion for numeric primitive values.

Note: Such a conversion will often result in the loss of data, and that loss is the responsibility of the programmer who performs the cast.

Assignment compatibility for references

Assignment compatibility for references doesn't involve range issues, as is the case with primitives. Rather, the reference to an object instantiated from a given class can be assigned to (is assignment-compatible with):

1. Any reference variable whose type is the same as the class from which the object was instantiated.
2. Any reference variable whose type is a superclass of the class from which the object was instantiated.
3. Any reference variable whose type is an interface that is implemented by the class from which the object was instantiated.
4. Any reference variable whose type is an interface that is implemented by a superclass of the class from which the object was instantiated.
5. A couple of other cases involving interfaces that extend other interfaces.

In this module, we are interested only in cases 1 and 2 above. We will be interested in the other cases in future modules involving interfaces.

Such an assignment does not require the use of a cast operator.

Type **Object** is completely generic

As mentioned earlier, a reference to any object can be assigned to a reference variable of the type **Object** , because the **Object** class is a superclass of every other class.

Converting reference types with a cast

Assignments of references, other than those listed above, require the use of a cast operator to purposely change the type of the reference.

However, it is not possible to perform a successful cast to convert the type of a reference to another type in all cases.

Generally, a cast can only be performed among reference types that fall on the same ancestral line of the class hierarchy, or on an ancestral line of an interface hierarchy. For example, a reference cannot be successfully cast to the type of a sibling or a cousin in the class hierarchy.

Downcasting

When we cast a reference along the class hierarchy in a direction away from the *root* class **Object** toward the *leaves* , we often refer to it as a *downcast* .

While it is also possible to cast in the direction from the *leaves* to the *root* , this conversion happens automatically, and the use of a cast operator is not required.

Preview

A sample program is provided that illustrates much of the detail involved in type conversion, method invocation, and casting with respect to reference

types.

Discussion and sample code

The program named **Polymorph02** , shown in [Listing 11](#) near the end of the module illustrates the use of the cast operator with references.

When you examine that program, you will see that two classes named **A** and **C** each extend the class named **Object** . Hence, we might say that they are siblings in the class hierarchy.

Another class named **B** extends the class named **A** . Thus, we might say that **A** is a child of **Object** , and **B** is a child of **A** .

The class named A

The definition of the class named **A** is shown in [Listing 1](#). This class implicitly extends the class named **Object** by default.

Note:

Listing 1 . Source code for class A.

```
using System;

class A {
    //this class is empty
} //end class A
```

The class named **A** is empty. It was included in this example for the sole purpose of adding a layer of inheritance to the class hierarchy.

The class named **B**

[Listing 2](#) shows the definition of the class named **B** . This class extends the class named **A** as indicated by the colon that joins the **B** and the **A** on the first line.

Note:

Listing 2 . Source code for class B.

```
class B : A {  
    public void m() {  
        Console.WriteLine("m in class B");  
    } //end method m()  
} //end class B
```

The method named **m**

The class named **B** defines a method named **m** . The behavior of the method is simply to display a message each time it is called.

The class named **C**

[Listing 3](#) shows the definition of the class named **C** , which also extends **Object** by default.

Note:

Listing 3 . Source code for class C.

```
class C {  
    //this class is empty  
} //end class C
```


The class named **C** is also empty. It was included in this example as a sibling class for the class named **A** . Stated differently, it was included as a class that is not in the ancestral line of the class named **B** .

The driver class

[Listing 4](#) shows the beginning of the driver class named **Polymorph02** , which also extends **Object** by default.

Note:

Listing 4 . Beginning of the class named Polymorph02.

```
public class Polymorph02 {  
    public static void Main() {  
        Object var = new B();  
    }  
}
```

An object of the class named B

This code instantiates an object of the class **B** and assigns the object's reference to a reference variable of type **Object** .

Note: Let me repeat what I just said for emphasis. The reference to the object of type **B** was not assigned to a reference variable of type **B** . Instead, it was assigned to a reference variable of type **Object** .

This assignment is allowable because **Object** is a superclass of **B** . In other words, the reference to the object of the class **B** is *assignment-compatible* with a reference variable of the type **Object** .

Automatic type conversion

In this case, the reference of type **B** is automatically converted to type **Object** and assigned to the reference variable of type **Object** .

Note: Note that the use of a cast operator was not required in this assignment.

Only part of the story

However, assignment compatibility is only part of the story. The simple fact that a reference is assignment-compatible with a reference variable of a given type says nothing about what can be done with the reference after it is assigned to the reference variable.

An illegal operation

For example, in this case, the reference variable that was automatically converted to type **Object** cannot be used directly to call the method named **m** on the object of type **B**. This is indicated in [Listing 5](#).

Note:

Listing 5 . Try to call method **m** on variable **var**.

```
//Following will not compile
//var.m();
```

A compiler error

An attempt to call the method named **m** on the reference variable of type **Object** in [Listing 5](#) resulted in the following compiler error:

```
error CS0117:  
'object' does not contain a definition for 'm'
```

It was necessary to convert the statement to a comment in order to cause the program to compile successfully.

An important rule

In order to use a reference of a class type to call a method, the method must be defined at or above that class in the class hierarchy. Stated differently, the method must either be defined in, or inherited into that class.

This case violates the rule

In this case, the method named **m** is defined in the class named **B** , which is two levels down from the class named **Object** .

Note: The method named **m** is neither defined in nor inherited into the class named **Object** .

When the reference to the object of the class **B** was assigned to the reference variable of type **Object** , the type of the reference was automatically converted to type **Object** .

Therefore, because the reference is type **Object** , it cannot be used directly to call the method named **m** .

The solution is a downcast

In this case, the solution to the problem is a downcast.

The code in [Listing 6](#) shows an attempt to solve the problem by casting the reference down the hierarchy to type **A** .

Note:

Listing 6 . Try a downcast to type A.

```
//Following will not compile  
//((A)var).m();
```

Still doesn't solve the problem

However, this still doesn't solve the problem, and the result is another compiler error.

Note: The method named **m** is neither defined in nor inherited into the class named **A** .

Again, it was necessary to convert the statement into a comment in order to cause the program to compile.

What is the problem here?

The problem with this approach is that the downcast simply didn't go far enough down the inheritance hierarchy.

The class named **A** does not contain a definition of the method named **m** . Neither does it inherit the method named **m** . The method named **m** is defined in class **B**, which is a subclass of **A** .

Therefore, a reference of type **A** is no more useful than a reference of type **Object** insofar as calling the method named **m** is concerned.

The real solution

The solution to the problem is shown in [Listing 7](#).

Note:

Listing 7 . Try a downcast to type B.

```
//Following will compile and run  
((B)var).m();
```

The code in [Listing 7](#) casts (temporarily converts) the reference value contained in the **Object** variable named **var** to type **B** .

The method named **m** is defined in the class named **B** . Therefore, a reference of type **B** can be used to call the method.

The code in [Listing 7](#) compiles and executes successfully. This causes the method named **m** to execute, producing the following output on the computer screen:

```
m in class B
```

A few odds and ends

Before leaving this topic, let's look at a few more issues.

The code in [Listing 8](#) declares and populates a new variable of type **B**.

Note:

Listing 8 . Assign var to v1.

```
//Following will compile and run  
B v1 = (B)var;
```

The code in [Listing 8](#) uses a cast to:

- Convert the contents of the **Object** variable to type **B**
- Assign the converted reference to the new reference variable of type **B**
-

A legal operation

This is a legal operation. In this class hierarchy, the reference to the object of the class **B** can be assigned to a reference variable of the types **B**, **A** , or **Object** .

Note: While this operation is legal, it is usually not a good idea to have two different reference variables that contain references to the same object. In this case, the variables named **var** and **v1** both contain a reference to the same object.

Cannot be assigned to type C

However, the reference to the object of the class **B** cannot be assigned to a reference variable of any other type, including type **C** . An attempt to do so is shown in [Listing 9](#).

Note:

Listing 9 . Cannot be assigned to C.

```
//Following will not execute.  
// Causes a runtime exception.  
//C v2 = (C)var;
```

The code in [Listing 9](#) attempts to cast the reference to type **C** and assign it to a reference variable of type **C** .

A runtime exception

Although the program will compile, it won't execute. An attempt to execute the statement in [Listing 9](#) results in an exception at runtime.

As a result, it was necessary to convert the statement into a comment in order to execute the program.

Another failed attempt

Similarly, an attempt to cast the reference to type **B** and assign it to a reference variable of type **C** , as shown in [Listing 10](#), won't compile.

Note:

Listing 10 . Another failed attempt.

```
//Following will not compile
//C v3 = (B)var;

//Pause until user presses any key.
Console.ReadKey();
} //end Main
} //end class Polymorph02
```

The problem here is that the class **C** is not a superclass of the class named **B** . Therefore, a reference of type **B** is not assignment-compatible with a reference variable of type **C** .

Again, it was necessary to convert the statement to a comment in order to compile the program.

The end of the program

[Listing 10](#) signals the end of the **Main** method, the end of the class, and the end of the program.

Run the program

I encourage you to copy the code from [Listing 11](#) . Use that code to create a C# console project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Run my program

Click [here](#) to download a zip file containing my version of the program. Extract the folder named **Polymoeph02** from the zip file and save it somewhere on your disk. Start **Visual C# 2010 Express** and select **Open Project...** from the **File** menu. Navigate to the project folder and select the file with the extension of **.sln** . This should cause the project to open and be ready to run or debug as described in the earlier module titled [Getting Started](#).

Summary

This module discusses type conversion for primitive and reference types.

A value of a particular type may be assignment-compatible with variables of other types.

If the type of a value is not assignment-compatible with a variable of a given type, it may be possible to perform a cast on the value to change its type and assign it to the variable as the new type. For primitive types, this will often result in the loss of information.

In general, numeric values of primitive types can be assigned to any variable whose type represents a numeric range that is as wide as or wider than the range of the value's type. (Values of type **bool** can only be assigned to variables of type **bool** .)

With respect to reference types, the reference to an object instantiated from a given class can be assigned to any of the following without the use of a cast:

- Any reference variable whose type is the same as the class from which the object was instantiated.
- Any reference variable whose type is a superclass of the class from which the object was instantiated.
- Any reference variable whose type is an interface that is implemented by the class from which the object was instantiated.
- Any reference variable whose type is an interface that is implemented by a superclass of the class from which the object was instantiated.

- A couple of other cases involving interfaces that extend other interfaces.

Assignments of references, other than those listed above, require the use of a cast to change the type of the reference.

It is not always possible to perform a successful cast to convert the type of a reference. Whether or not a cast can be successfully performed depends on the relationship of the classes involved in the class hierarchy.

A reference to any object can be assigned to a reference variable of the type **Object** , because the **Object** class is a superclass of every other class.

When we cast a reference along the class hierarchy in a direction away from the root class **Object** toward the leaves, we often refer to it as a *downcast* .

Whether or not a method can be called on a reference to an object depends on the current type of the reference and the location in the class hierarchy where the method is defined. In order to use a reference of a class type to call a method, the method must be defined in or inherited into that class.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Xna0112-Type Conversion, Casting, and Assignment Compatibility
- File: Xna0112.htm
- Published: 02/27/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the C# program discussed in this module is provided in [Listing 11](#).

Note:

Listing 11 . Project Polymorph02.

```
/*Project Polymorph02  
Copyright 2002, R.G.Baldwin
```

```
This program illustrates downcasting  
Program output is:
```

```
m in class B
```

```
*****
```

```
*****/
```

```
using System;
```

```

class A {
    //this class is empty
} //end class A
//=====
=====//

class B : A {
    public void m() {
        Console.WriteLine("m in class B");
    } //end method m()
} //end class B
//=====
=====//

class C {
    //this class is empty
} //end class C
//=====
=====//

public class Polymorph02 {
    public static void Main() {
        Object var = new B();
        //Following will not compile
        //var.m();
        //Following will not compile
        //((A)var).m();
        //Following will compile and run
        ((B)var).m();

        //Following will compile and run
        B v1 = (B)var;
        //Following will not execute.
        // Causes a runtime exception.
        //C v2 = (C)var;
        //Following will not compile
    }
}

```

```
//C v3 = (B)var;  
  
    //Pause until user presses any key.  
    Console.ReadKey();  
} //end Main  
} //end class Polymorph02  
//=====
```

```
=====/  

```

-end-

Xna0114-Runtime Polymorphism through Class Inheritance

With runtime polymorphism, the selection of a method for execution is based on the actual type of the object whose reference is stored in a reference variable, and not on the type of the reference variable on which the method is called.

Revised: Fri May 06 15:50:20 CDT 2016

This page is part of a Book titled [XNA Game Studio](#).

Table of Contents

- [Table of Contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Listings](#)
- [General background information](#)
 - [What is polymorphism?](#)
 - [How does C# implement polymorphism?](#)
 - [The essence of runtime polymorphic behavior](#)
 - [The decision process](#)
 - [Late binding](#)
 - [Operational description of runtime polymorphism](#)
 - [Runtime polymorphism is very powerful](#)
 - [An important attribute of runtime polymorphism](#)
 - [Why is it called runtime polymorphism?](#)
 - [Why defer the decision?](#)
 - [Could be either type](#)
- [Preview](#)
- [Discussion and sample code](#)

- The class named A
 - A virtual method
 - Behavior of the virtual method
- The class named B
 - The override declaration
 - A compiler warning
 - Overriding versus hiding
 - Behavior of the overridden method
- The driver class
 - A new object of the class B
 - Downcast and call the method
 - Which version was executed?
 - Why was this version executed?
 - Not runtime polymorphic behavior
 - This is runtime polymorphic behavior
 - The method output
 - Very important
 - Another invocation of the method
 - Compiler error
- Some important rules
 - Necessary, but not sufficient
 - Must define or inherit the method
- One additional scenario
 - A new object of type A
 - Downcast and call the method

- [The output](#)
- [Not polymorphic behavior](#)
- [Once again, what is runtime polymorphism?](#)
- [Run the program](#)
- [Run my program](#)
- [Summary](#)
- [Miscellaneous](#)
- [Complete program listing](#)

Preface

This module is one in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

An earlier module titled [Getting Started](#) provided information on how to get started programming with Microsoft's XNA Game Studio.

Object-oriented programs exhibit three main characteristics:

- Encapsulation
- Inheritance
- Polymorphism

I have explained encapsulation, inheritance, and compile-time polymorphism in earlier modules. I will continue my explanation of polymorphism in this module with an explanation of runtime polymorphism using method overriding and class inheritance. I will defer an explanation of polymorphism using interface inheritance until a future module.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Listings while you are reading about them.

Listings

- [Listing 1](#). Class A.
- [Listing 2](#). Class B.
- [Listing 3](#). Beginning of the driver class.
- [Listing 4](#). This is runtime polymorphic behavior.
- [Listing 5](#). A failed attempt.
- [Listing 6](#). Not polymorphic behavior.
- [Listing 7](#). Project Polymorph03.

General background information

What is polymorphism?

As you learned in an earlier module, the meaning of the word polymorphism is something like *one name, many forms* .

How does C# implement polymorphism?

Also as you learned in an earlier module, polymorphism manifests itself in C# in the form of multiple methods having the same name.

In some cases, multiple methods have the same name, but different formal argument lists. This is polymorphism implemented using *overloaded* methods. I explained this form of polymorphism in an earlier module.

In other cases, multiple methods have the same name, same return type, and same formal argument list. This is method *overriding* , which is the main topic of this module.

The essence of runtime polymorphic behavior

Methods are called on references to objects. Typically, those references are stored in reference variables, or in the elements of a collection such as an array, stack, or queue.

The decision process

With runtime polymorphism based on *method overriding*, the decision regarding which version of a method will be executed is based on the actual **type of the object** whose reference is stored in the reference variable, and **not** on the **type of the reference variable** on which the method is called.

Stated differently, the type of the reference determines which methods **can be called**. The type of the object determines which method (from that set of allowable methods) **will be called**.

Late binding

The decision regarding which version of the method to call cannot be made at compile time. That decision must be deferred and made at runtime. This is sometimes referred to as *late binding*.

Operational description of runtime polymorphism

Here is an operational description of runtime polymorphism as implemented in C# through inheritance and method overriding:

- Assume that a class named **SuperClass** defines a method named **method**.
- Assume that a class named **SubClass** extends **SuperClass** and overrides the method named **method**.
- Assume that a reference to an object of the class named **SubClass** is assigned to a reference variable named **ref** of type **SuperClass**.

- Assume that the method named **method** is then called on the reference variable using the following syntax: **ref.method()** .
- Result: The version of the method named **method** that will actually be executed is the overridden version in the class named **SubClass** . The version that is defined in the class named **SuperClass** will not be not executed.

This is runtime polymorphism.

Runtime polymorphism is very powerful

As you gain more experience with C#, you will learn that much of the power of OOP using C# is centered on runtime polymorphism using class inheritance, interfaces, and method overriding. *(The use of interfaces for polymorphism will be discussed in a subsequent module.)*

An important attribute of runtime polymorphism

This is worth repeating:

Note: *The decision regarding which version of the method to execute is based on the actual type of object whose reference is stored in the reference variable, and not on the type of the reference variable on which the method is called.*

Why is it called runtime polymorphism?

The reason that this type of polymorphism is often referred to as *runtime polymorphism* is because the decision regarding which version of the method to execute cannot be made until runtime. The decision cannot be made at compile time *(as is the case with overloaded methods)*.

Why defer the decision?

The decision cannot be made at compile time because the compiler has no way of knowing (*when the program is compiled*) the actual type of the object whose reference will be stored in the reference variable.

For example, the type of the object might be the result of a choice made at runtime by a human user among several different possible choices.

Could be either type

For the situation described earlier, that object could just as easily be of type **SuperClass** as of type **SubClass** . In either case, it would be valid to assign the object's reference to the same superclass reference variable.

If the object were of the **SuperClass** type, then a call to the method named **method** on the reference would cause the version of the method defined in **SuperClass** , and not the version defined in **SubClass** , to be executed.

Note: *One more time -- the version that is executed is determined by the type of the object and not by the type of the reference variable containing the reference to the object.*

Preview

From a practical programming viewpoint, polymorphism manifests itself in three distinct forms in C#:

- Method overloading
- Method overriding through class inheritance
- Method overriding through the C# interface inheritance

I covered method *overloading* as one form of polymorphism (*compile-time polymorphism*) in an earlier module.

I will begin the discussion of runtime polymorphism through method *overriding* and class inheritance in this module. I will cover interfaces in a future module.

Discussion and sample code

Let's examine a sample program that illustrates runtime polymorphism using class inheritance and overridden methods. The name of the program is **Polymorph03** . A complete listing of the program is provided in [Listing 7](#) near the end of the module.

The class named A

I will discuss this program in fragments. [Listing 1](#) shows the definition of a class named **A** , which extends the class named **Object** by default.

Note:

Listing 1 . Class A.

```
using System;

class A {
    public virtual void m() {
        Console.WriteLine("m in class A");
    } //end method m()
} //end class A
```

The class named **A** defines a simple method named **m** .

A virtual method

Note that the method named **m** is declared to be **virtual** . This means that it is allowable to override this method in a subclass.

Note: If you come from a Java programming background, you will note that this is the reverse of the situation in Java. In Java, all methods are virtual by default unless they are declared to be final.

Behavior of the virtual method

The behavior of the virtual method, as defined in the class named **A** , is to display a message indicating that it has been called, and that it is defined in the class named **A** .

This message will allow us to determine which version of the method is executed in each case discussed later.

The class named B

[Listing 2](#) shows the definition of a class named **B** that extends the class named **A**.

Note:

Listing 2 . Class B.

```
class B : A {  
    public override void m() {  
        Console.WriteLine("m in class B");  
    }//end method m()  
}//end
```

The class named **B** overrides (*redefines*) the method named **m** , which it inherits from the class named **A** .

The override declaration

Note the use of the **override** declaration in [Listing 2](#). In C#, if one method overrides another, it is necessary to declare that fact.

Note: Once again, if you come from a Java background, you will note that this is just the reverse of the situation in Java. In Java, a method whose name, return type, and formal argument list matches an inherited method will automatically override the inherited method.

A compiler warning

If you fail to make the override declaration in [Listing 2](#), you will get a compiler warning that reads something like the following:

Note:

```
warning CS0114: 'B.m()' hides inherited member 'A.m()'.
```

```
To make the current member override that implementation, add the override keyword. Otherwise add the new keyword.
```

Overriding versus hiding

I'm not going to get into a discussion of the difference between overriding and hiding in this module. Perhaps I will find the time to provide such a discussion in a future module.

Behavior of the overridden method

Like the inherited version of the method, the overridden version displays a message indicating that it has been called. However, the message is different from the message displayed by the inherited version discussed above. The overridden version tells us that it is defined in the class named **B**.

Note: *According to the current jargon, the behavior of the overridden version of the method is appropriate for an object instantiated from the class named **B**.*

Again, this message will allow us to determine which version of the method is executed in each case discussed later.

The driver class

[Listing 3](#) shows the beginning of the driver class named **Polymorph03**.

Note:

Listing 3. Beginning of the driver class.

```
public class Polymorph03 {  
    public static void Main() {
```



```
Object var = new B();  
//Following will compile and run  
((B)var).m();
```

A new object of the class B

The code in the **Main** method begins by instantiating a new object of the class named **B** , and assigning the object's reference to a reference variable of type **Object** .

Note: This is legal because an object's reference can be assigned to any reference variable whose type is a superclass of the class from which the object was instantiated. The class named **Object** is the superclass of all classes.

Downcast and call the method

If you have read the module titled [Xna0112-Type Conversion, Casting, and Assignment Compatibility](#), it will come as no surprise to you that the second statement in the **Main** method, which casts the reference down to type **B** and calls the method named **m** on it, will compile and execute successfully.

Which version was executed?

The execution of the method produces the following output on the computer screen:

m in class B

By examining the output, you can confirm that the version of the method that was overridden in the class named **B** is the version that was executed.

Why was this version executed?

This should also come as no surprise to you. The cast converts the type of the reference from type **Object** to type **B**.

You can always call a public method belonging to an object using a reference to the object whose type is the same as the class from which the object was instantiated.

Not runtime polymorphic behavior

Just for the record, the above invocation of the method does not constitute runtime polymorphism (*in my opinion*). I included that invocation of the method to serve as a backdrop for what follows.

Note: If this is runtime polymorphism, it is not a very significant example, because there was no requirement for the runtime system to decide between different methods having the same name. The runtime system simply executed a method belonging to an object using a reference of the same type as the object.

This is runtime polymorphic behavior

However, the call to the method in [Listing 4](#) does constitute runtime polymorphism.

Note:

Listing 4 . This is runtime polymorphic behavior.

```
//Following will also compile  
// and run due to polymorphic  
// behavior.  
((A)var).m();
```

The statement in [Listing 4](#) casts the reference down to type **A** and calls the method named **m** on that reference.

The method output

Here is the punch line. Not only does the statement in [Listing 4](#) compile and run successfully, it produces the following output, (*which is exactly the same output as before*) :

m in class B

The same method was executed in both cases

Very important

It is very important to note that this output, (*produced by casting the reference variable to type **A** instead of type **B***) , is exactly the same as that produced by the earlier call to the method when the reference was cast to type **B** . This means that the same version of the method was executed in both cases.

This confirms that even though the type of the reference was converted to type **A** , (*rather than type **Object** or type **B***) , the overridden version of the method defined in class **B** was actually executed.

This is runtime polymorphic behavior in a nutshell.

The version of the method that was executed was based on the **actual type of the object, B** , and **not** on the **type of the reference, A** . This is an extremely powerful and useful concept.

Another invocation of the method

Now take a look at the statement in [Listing 5](#) . Will this statement compile and execute successfully? If so, which version of the method will be executed?

Note:

Listing 5 . A failed attempt.

```
//Following will not compile  
//var.m();
```

(That was easy. The answer to the question is given in [Listing 5](#)) .

Compiler error

The code in [Listing 5](#) attempts, unsuccessfully, to call the method named **m** using the reference variable named **var** , which is of type **Object** . The result is a compiler error, which reads something like the following:

Note:

```
error CS0117: 'object' does not contain a  
definition for 'm'
```

Some important rules

The **Object** class does not define a method named **m** . Therefore, the overridden method named **m** in the class named **B** is not an overridden version of a method that is defined in the class named **Object** .

Necessary, but not sufficient

Runtime polymorphism based on class inheritance requires that the type of the reference variable be a superclass of the class from which the object (*on which the method will be called*) is instantiated.

Note: At least this requirement is true if a significant decision among methods is to be made.

However, while necessary, that is not sufficient to ensure runtime polymorphic behavior.

Must define or inherit the method

The type of the reference variable must also be the name of a class that either defines or inherits a version of the method that will ultimately be called on the object.

Since the class named **Object** does not define (*or inherit*) the method named **m** , a reference of type **Object** does not qualify as a participant in runtime polymorphic behavior in this case. The attempt to use it as a participant results in the compiler error given above.

One additional scenario

Before leaving this topic, let's look at one additional scenario to help you distinguish what is, and what is not, runtime polymorphism. Consider the code shown in [Listing 6](#).

Note:

Listing 6 . Not polymorphic behavior.

```
//Instantiate obj of class A
var = new A();
//Call the method on it
((A)var).m();

// Pause until the user presses any key.
Console.ReadKey();
} //end Main
} //end class Polymorph03
```

A new object of type A

The code in [Listing 6](#) instantiates a new object of the class named **A** , and stores the object's reference in the original reference variable named **var** of type **Object** .

Note: As a side note, this overwrites the previous contents of the reference variable with a new reference and causes the object whose reference was previously stored there to become eligible for garbage collection.

Downcast and call the method

Then the code in [Listing 6](#) casts the reference down to type **A** , (*the type of the object to which the reference refers*) , and calls the method named **m** on the downcast reference.

The output

As you would probably predict, this produces the following output on the computer screen:

```
m in class A
```

In this case, the version of the method defined in the class named **A** , (*not the version defined in B*) was executed.

Not polymorphic behavior

Once again, in my view, this is not runtime polymorphic behavior (*at least it isn't a very useful form of polymorphic behavior*) . This code simply converts the type of the reference from type **Object** to the type of the class from which the object was instantiated, and calls one of its methods. Nothing special takes place regarding a selection among different versions of the method.

Once again, what is runtime polymorphism?

As I have discussed in this module, runtime polymorphic behavior based on inheritance occurs when

- The type of the reference is a superclass of the class from which the object was instantiated.
- The version of the method that is executed is the version that is either defined in, or inherited into, the class from which the object was instantiated.

And that is probably more than you ever wanted to hear about runtime polymorphism based on inheritance.

A future module will discuss runtime polymorphism based on the C# interface. From a practical viewpoint, you will find the rules to be similar but somewhat different in the case of the C# interface.

Run the program

I encourage you to copy the code from [Listing 7](#). Use that code to create a C# console project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Run my program

Click [here](#) to download a zip file containing my version of the program. Extract the folder named **Polymorph03** from the zip file and save it somewhere on your disk. Start **Visual C# 2010 Express** and select **Open Project...** from the **File** menu. Navigate to the project folder and select the file with the extension of **.sln**. This should cause the project to open and be ready to run or debug as described in the earlier module titled [Getting Started](#).

Summary

Polymorphism manifests itself in C# in the form of multiple methods having the same name.

From a practical programming viewpoint, polymorphism manifests itself in three distinct forms in C#:

- Method overloading
- Method overriding through class inheritance
- Method overriding through interface inheritance

This module discusses method overriding through class inheritance.

With runtime polymorphism based on method overriding, the decision regarding which version of a method will be executed is based on the actual type of object whose reference is stored in a reference variable, and not on the type of the reference variable on which the method is called.

The decision regarding which version of the method to call cannot be made at compile time. That decision must be deferred and made at runtime. This is sometimes referred to as late binding.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Xna0114-Runtime Polymorphism through Class Inheritance
- File: Xna0114.htm
- Published: 02/27/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available

on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the program discussed in this module is provided in [Listing 7](#).

Note:

Listing 7 . Project Polymorph03.

```
/*Project Polymorph03
Copyright 2009, R.G.Baldwin
```

This program illustrates downcasting
and polymorphic behavior

Program output is:

```
m in class B
m in class B
m in class A
*****
*****/
using System;

class A {
    public virtual void m() {
        Console.WriteLine("m in class A");
    } //end method m()
} //end class A
//=====
```

```

=====//

class B : A {
    public override void m() {
        Console.WriteLine("m in class B");
    }//end method m()
}//end class B
//=====
=====//

public class Polymorph03 {
    public static void Main() {
        Object var = new B();
        //Following will compile and run
        ((B)var).m();
        //Following will also compile
        // and run due to polymorphic
        // behavior.
        ((A)var).m();
        //Following will not compile
        //var.m();
        //Instantiate obj of class A
        var = new A();
        //Call the method on it
        ((A)var).m();

        // Pause until the user presses any key.
        Console.ReadKey();
    }//end Main
}//end class Polymorph03

```

-end-

Xna0116-Runtime Polymorphism and the Object Class

Baldwin explains the use of the Object class as a completely generic type for storing references to objects of subclass types, and explains how that results in a very useful form of runtime polymorphism.

Revised: Sat May 07 11:13:14 CDT 2016

This page is part of a Book titled [XNA Game Studio](#).

Table of Contents

- [Table of Contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [The generic type Object](#)
 - [References of type Object](#)
 - [Methods in the Object class](#)
 - [The difference between string and String](#)
 - [Every class inherits these methods](#)
 - [To be overridden...](#)
 - [Calling methods of the Object class](#)
 - [And the behavior will be...](#)
- [Preview](#)
- [Discussion and sample code](#)
 - [The class named A](#)
 - [Does not override the ToString method](#)
 - [The class named B](#)

- Overrides the ToString method
 - Purpose of the ToString method
 - Can be overridden
 - Behavior of the default and overridden versions
 - Will be useful later
- The class named C
 - Behavior of overridden version
- The driver class
 - A new object of the class A
 - Call ToString method on the reference
 - Display the returned String
 - Default ToString behavior
 - Class A does not override ToString
 - A new object of the class B
 - Call ToString and display the result
 - Do you recognize this?
 - Overridden version of ToString was executed
 - Once again, what is the rule?
 - An object of the class C
 - What will the output look like?
 - Overridden version of ToString was called
- No downcasting was required
- A generic array object
- Collections
- Run the program
- Run my_program
- Summary
- Miscellaneous

- [Complete program listing](#)

Preface

This module is one in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

An earlier module titled [Getting Started](#) provided information on how to get started programming with Microsoft's XNA Game Studio.

Object-oriented programs exhibit three main characteristics:

- Encapsulation
- Inheritance
- Polymorphism

I have explained encapsulation, inheritance, compile-time polymorphism, and runtime polymorphism using method overriding and class inheritance in earlier modules. This module will explain the importance of the **Object** class in polymorphic behavior. I will defer an explanation of polymorphism using interface inheritance until a future module.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Text output for ToString method and class A.
- [Figure 2](#). Text output for overridden ToString method and class B.
- [Figure 3](#). Text output for overridden ToString method and class C.

Listings

- [Listing 1](#). Definition of class A.
- [Listing 2](#). Definition of class B.
- [Listing 3](#). Definition of class C.
- [Listing 4](#). Beginning of the driver class.
- [Listing 5](#). A new object of the class B.
- [Listing 6](#). An object of the class C.
- [Listing 7](#). Project Polymorph04.

General background information

The generic type Object

In this module, I will explain the use of the **Object** class as a completely generic type for storing references to objects of subclass types, and will explain how that results in a very useful form of runtime polymorphism.

I will briefly discuss the default versions of some of the methods defined in the **Object** class, and will explain that in many cases, those default versions are meant to be overridden.

References of type Object

The **Object** type is a completely generic type that can be used to store a reference to any object that can be instantiated in C#.

Methods in the Object class

The **Object** class defines the following [eight methods](#), which are inherited by every other class:

- **Equals** - Overloaded. Determines whether two **Object** instances are equal.

- **Equals(Object)** - Determines whether the specified **Object** is equal to the current **Object** .
- **Equals(Object, Object)** - Determines whether the specified **Object** instances are considered equal.
- **Finalize** - Allows an **Object** to attempt to free resources and perform other cleanup operations before the **Object** is reclaimed by garbage collection.
- **GetHashCode** - Serves as a hash function for a particular type.
- **GetType** - Gets the Type of the current instance.
- **MemberwiseClone** - Creates a shallow copy of the current **Object** .
- **ReferenceEquals** - Determines whether the specified **Object** instances are the same instance.
- **ToString** - Returns a **String** that represents the current **Object** .

Because these eight methods are inherited by every other class, they are always available for you to use in your code. *(Possibly the most frequently used of these methods is the **ToString** method.)*

Two of the methods in this list are defined in overloaded versions *(same name, different formal argument lists)* .

The difference between string and String

This [author](#) says "string is an alias for System.String. So technically, there is no difference. It's like int vs. System.Int32."

Every class inherits these methods

Because every class is either a direct or indirect subclass of **Object** , every class in C#, *(including new classes that you define)* , inherit these methods.

To be overridden...

Some of these methods are intended to be overridden for various purposes. This includes **Equals** , **Finalize** , **GetHashCode** , and **ToString** , which are all declared **virtual** .

However, some of them, such as **GetType** , are not declared **virtual** and therefore are intended to be used directly without overriding.

Calling methods of the **Object** class

You can store a reference to any object in a reference variable of type **Object** .

Because every new class inherits these methods, you can call any of these methods on any reference to any object stored in a reference variable of type **Object** or in a reference variable of any other type.

And the behavior will be...

If the class from which the object was instantiated inherits or defines an overridden version of one of the methods in the above list, calling that method on the reference will cause the overridden version to be executed.

Otherwise, calling that method on the reference will cause the default version defined in the **Object** class to be executed.

Preview

The behavior described above is illustrated in the sample program named **Polymorph04** , which you can view in its entirety in [Listing 7](#) near the end of this module.

For purposes of illustration, this program deals specifically with the method named **ToString** from the above list, but could deal with the other virtual methods in the list as well.

Discussion and sample code

The class named A

[Listing 1](#) defines a class named **A**, which explicitly extends the class named **Object** .

*(Recall that classes extend **Object** by default. It is not necessary to explicitly show that a class extends **Object** . I showed that here simply to remind you that all classes in C# are rooted in the class named **Object** .)*

Note:

Listing 1 . Definition of class A.

```
using System;

class A : Object {
    //This class is empty
} //end class A
```

Does not override the ToString method

The most important thing to note about the class named **A** is that it does not override any of the methods that it inherits from the class named **Object** .

Therefore, it inherits the default version of the method named **ToString** from the class named **Object** .

(We will see an example of the behavior of the default version of that method shortly.)

The class named B

[Listing 2](#) defines the class named **B** . This class extends the class named **A** .

Note:

Listing 2 . Definition of class B.

```
class B : A {  
    public override String ToString() {  
        return "ToString in class B";  
    }//end overridden ToString()  
}//end class B
```

Overrides the ToString method

Of particular interest, (*for purposes of this module*) , is the fact that the class named **B** overrides the inherited **ToString** method.

It inherits the default version of the **ToString** method, because its superclass named **A** , which extends **Object** , does not override the **ToString** method.

Purpose of the ToString method

The purpose of the **ToString** method is to return a reference to an object of the class **String** that represents an object instantiated from a class that overrides the method.

Here is part of what Microsoft has to say about the **ToString** method:

Note:

"This method returns a human-readable string that is culture-sensitive. For example, for an instance of the **Double** class whose value is zero, the

implementation of **Double.ToString** might return "0.00" or "0,00" depending on the current UI culture."

Can be overridden

The **ToString** method can be overridden in a subclass to return values that are meaningful for that type. Again, according to Microsoft:

Note:

"For example, the base data types, such as Int32, implement **ToString** so that it returns the string form of the value that the object represents."

Behavior of the default and overridden versions

The default implementation of the **ToString** method, as defined in the **Object** class, returns the fully qualified name of the type of the object.

I didn't override the **ToString** method in the class named **A** , but I did override it in the class named **B** , which is a subclass of **A** .

The behavior of my overridden version of the method in the class named **B** returns a reference to a **String** object, containing text that indicates that the overridden version of the method in the class named **B** has been executed.

Will be useful later

The reference to the **String** object returned by the overridden version of the method will prove useful later when we need to determine which version of the method is actually executed.

The class named C

[Listing 3](#) shows the definition of a class named **C**, which extends the class named **B**, and overrides the method named **ToString** again.

An inherited virtual method can be overridden by every class that inherits it, resulting in potentially many different overridden versions of a given method in a class hierarchy.

Note:

Listing 3 . Definition of class C.

```
class C : B {  
    public override String ToString() {  
        return "ToString in class C";  
    } //end overridden ToString()  
} //end class B
```

Behavior of overridden version

The behavior of this overridden version of the method is similar to, but different from the overridden version in the class **B**.

In this case, the method returns a reference to a **String** object that can be used to confirm that this overridden version of the method has been executed.

The driver class

[Listing 4](#) shows the beginning of the driver class named **Polymorph04**.

Note:

Listing 4 . Beginning of the driver class.

```
public class Polymorph04 {  
    public static void Main() {  
        Object varA = new A();  
        String v1 = varA.ToString();  
        Console.WriteLine(v1);  
    }  
}
```

A new object of the class A

The **Main** method of the driver class begins by instantiating a new object of the class **A**, and saving the object's reference in a reference variable of type **Object** , named **varA** .

It is very important to note that even though the object was instantiated from the class named **A** , the reference was saved as type **Object** , not type **A**.

Call ToString method on the reference

Then the code in [Listing 4](#) calls the **ToString** method on the reference variable named **varA** , saving the returned reference to the **String** object in a reference variable of type **String** named **v1** .

Display the returned String

Finally, that reference is passed to the **WriteLine** method, causing the **String** returned by the **ToString** method to be displayed on the computer screen. This causes the text shown in [Figure 1](#) to be displayed.

Note:

Figure 1 . Text output for ToString method and class A.

A

Default ToString behavior

What you are seeing here is the **String** produced by the default version of the **ToString** method, as defined by the class named **Object** . The default implementation of the **ToString** method returns the fully qualified name of the *type* of the object. In this case, the object was instantiated from the class named **A** (*so the type of the object is A*) .

Class A does not override ToString

Recall that our new class named **A** does not override the **ToString** method. Therefore, when the **ToString** method is called on a reference to an object of the class **A**, the default version of the method is executed, producing the output shown in [Figure 1](#).

A new object of the class B

Now consider the code shown in [Listing 5](#), which instantiates a new object of the class named **B** , and stores the object's reference in a reference variable of type **Object** .

Note:

Listing 5 . A new object of the class B.

```
Object varB = new B();  
String v2 = varB.ToString();
```

```
Console.WriteLine(v2);
```

Call `ToString` and display the result

The code in [Listing 5](#) calls the **ToString** method on the reference of type **Object**, saving the returned reference in the reference variable named **v2**. (Recall that the **ToString** method is overridden in the class named **B**.)

As before, the reference is passed to the **WriteLine** method, which causes the text shown in [Figure 2](#) to be displayed on the computer screen.

Note:

Figure 2. Text output for overridden `ToString` method and class **B**.

`ToString` in class **B**

Do you recognize this?

You should recognize this as the text that was encapsulated in the **String** object returned by the overridden version of the **ToString** method defined in the class named **B**. (See [Listing 2](#).)

Overridden version of `ToString` was executed

This verifies that even though the reference to the object of the class **B** was stored in a reference variable of type **Object**, the overridden version of the **ToString** method in the class named **B** was executed (*instead of the default version defined in the class named **Object***). This is **runtime polymorphic behavior**, as described in a previous module.

Once again, what is the rule?

The selection of a method for execution is based on the *actual type of object* whose reference is stored in a reference variable, and not on the *type of the reference variable* on which the method is called.

An object of the class C

Finally, the code in [Listing 6](#)

- Instantiates a new object of the class **C**
- Stores the object's reference in a reference variable of type **Object**
- Calls the **ToString** method on the reference and saves the returned string
- Displays the returned string on the computer screen

Note:

Listing 6 . An object of the class C.

```
Object varC = new C();
String v3 = varC.ToString();
Console.WriteLine(v3);

//Pause until user presses any key.
Console.ReadKey();
} //end Main
} //end class Polymorph04
```

What will the output look like?

By now, you should know what to expect in the way of text appearing on the computer screen. The code in [Listing 6](#) causes the text shown in [Figure 3](#) to be displayed.

Note:

Figure 3 . Text output for overridden ToString method and class C.

ToString in class C

Overridden version of ToString was called

This confirms what you should already have known by now. In particular, even though the reference to the object of the class **C** was stored in a reference variable of type **Object** , the overridden version of the **ToString** method defined in the class named **C** was executed. Again, this is *runtime polymorphic behavior* based on class inheritance and method overriding.

No downcasting was required

It is also very important to note that no downcasting was required in order to call the **ToString** method in any of the cases shown above.

Because a default version of the **ToString** method is defined in the **Object** class, that method can be called without a requirement for downcasting on a reference to any object stored in a variable of type **Object** . This holds true for any of the methods defined in the class named **Object** .

A generic array object

Therefore, if we create an array object designed to store references of type **Object** , we can store (*potentially mixed*) references to any type of object in that array. We can later call any of the methods defined in the **Object** class on any of the references stored in the array.

The result will be the execution of the overridden version of the method as defined in the class from which the object was instantiated, or the version

inherited into that class if the method is not overridden in that class. Current jargon would say that the behavior of the method is *appropriate* for the type of object on which it is called.

Collections

The C# library provides a number of generic data structure classes, such as **Stack** , and **Queue** , which store and retrieve references to objects as type **Object** . These classes can be used to create objects that can store and retrieve objects of any class.

Run the program

I encourage you to copy the code from [Listing 7](#) . Use that code to create a C# console project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Run my program

Click [here](#) to download a zip file containing my version of the program. Extract the folder named **Polymorph04** from the zip file and save it somewhere on your disk. Start **Visual C# 2010 Express** and select **Open Project...** from the **File** menu. Navigate to the project folder and select the file with the extension of **.sln** . This should cause the project to open and be ready to run or debug as described in the earlier module titled [Getting Started](#) .

Summary

Polymorphism manifests itself in C# in the form of multiple methods having the same name.

From a practical programming viewpoint, polymorphism manifests itself in three distinct forms in C#:

- Method overloading
- Method overriding through class inheritance
- Method overriding through interface inheritance

In this module, I have continued my discussion of the implementation of polymorphism using method overriding through class inheritance, and have concentrated on a special case in that category.

More specifically, I have discussed the use of the **Object** class as a completely generic type for storing references to objects of subclass types, and have explained how that results in a very useful form of runtime polymorphism.

I briefly mentioned the default version of the methods defined in the **Object** class, and explained that in many cases, those default versions are meant to be overridden.

I provided a sample program that illustrates the overriding of the **ToString** method, which is one of the methods defined in the **Object** class.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Xna0116-Runtime Polymorphism and the Object Class
- File: Xna0116.htm
- Published: 02/27/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it

possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the program discussed in this module is provided in [Listing 7](#).

Note:

Listing 7 . Project Polymorph04.

```
/*Project Polymorph04  
Copyright 2009, R.G.Baldwin
```

This program illustrates polymorphic behavior
and the Object class

Program output is:

```
A  
ToString in class B  
ToString in class C
```

```

*****
*****/
using System;

class A : Object {
    //This class is empty
} //end class A
//=====
=====//

class B : A {
    public override String ToString() {
        return "ToString in class B";
    } //end overridden ToString()
} //end class B
//=====
=====//

class C : B {
    public override String ToString() {
        return "ToString in class C";
    } //end overridden ToString()
} //end class B
//=====
=====//

public class Polymorph04 {
    public static void Main() {
        Object varA = new A();
        String v1 = varA.ToString();
        Console.WriteLine(v1);

        Object varB = new B();
        String v2 = varB.ToString();
        Console.WriteLine(v2);

        Object varC = new C();
    }
}

```

```
String v3 = varC.ToString();  
Console.WriteLine(v3);  
  
    //Pause until user presses any key.  
    Console.ReadKey();  
} //end Main  
} //end class Polymorph04
```

-end-

Xna0118-The XNA Framework and the Game Class

Use a very simple XNA program to learn many of the details regarding the incorporation of the XNA framework into the object-oriented C# programming language. Also learn about constructors, the this keyword, the base keyword, and some of the differences between a Console Application and a Windows Game application.

Revised: Sat May 07 18:29:12 CDT 2016

This page is part of a Book titled [XNA Game Studio](#).

Table of Contents

- [Table of Contents](#)
- [Preface](#)
 - [Moving from C# to XNA](#)
 - [My objective](#)
 - [Fasten your seatbelt](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [A software framework](#)
 - [What is a software framework?](#)
 - [New project choices in Visual C#](#)
 - [A Console Application](#)
 - [Start debugging](#)
 - [The Main method](#)
 - [The starting point for a Console Application](#)

- [A Windows Game Application](#)
 - [Source code files](#)
 - [Build and debug](#)
 - [The file named Program.cs](#)
 - [Instantiate a Game1 object and call the Run method](#)
 - [What do we know about the class named Game1?](#)
 - [The using directive](#)
 - [Memory management](#)
 - [Don't modify the file named Program.cs](#)
 - [The Game1 class](#)
 - [How to create a game](#)
 - [Game1 extends Game](#)
 - [Main method calls the Run method](#)
 - [The Run method](#)
 - [Override five methods](#)
 - [Initialization](#)
 - [The game loop](#)
 - [Override Update for game logic](#)
 - [Override the Draw method](#)
- [Preview](#)
 - [Program output](#)
 - [What if you don't honor the transparent background?](#)
 - [Not a very exciting program](#)
- [Discussion and sample code](#)
 - [Creating a new Windows Game project](#)
 - [Step 1: Create a new Windows Game project named XNA0118Proj](#)
 - [Select a Windows Game project](#)

- [Step 2: Add your image file to the Content folder](#)
 - [Add your image to the Content folder](#)
 - [The Asset Name](#)
- [Steps 3, 4, and 5: Write code](#)
 - [Modify two overridden methods](#)
 - [Will discuss in fragments](#)
- [Beginning of the class named Game1](#)
 - [The namespace](#)
- [General information](#)
 - [The superclass named Game](#)
 - [Overridden methods](#)
 - [The game loop](#)
 - [The big picture](#)
- [The constructor for the Game1 class](#)
 - [What is a constructor?](#)
 - [A new GraphicsDeviceManager object](#)
 - [Passing the this keyword as a parameter](#)
 - [The ContentManager](#)
 - [When the constructor in Listing 5 terminates...](#)
- [The Run method of the Game class](#)
 - [The BeginRun method](#)
- [The game loop](#)
- [The overridden LoadContent method](#)
 - [Two new instance variables](#)
 - [The Texture2D class](#)
 - [The SurfaceFormat enumeration](#)

- [The Texture Format of my image](#)
- [A new SpriteBatch object](#)
 - [The SpriteBatch class](#)
 - [This can be confusing](#)
 - [The inherited property](#)
 - [The new code](#)
 - [Generic methods](#)
 - [Required syntax for the Load method](#)
 - [What do the angle brackets mean?](#)
 - [Calling the Load method of the current ContentManager](#)
 - [Populate the variable named myTexture](#)
 - [The Vector2 structure](#)
- [The overridden Game.Draw method](#)
 - [Game loop timing](#)
- [More general information](#)
 - [What is a sprite?](#)
 - [What is a bitmap?](#)
 - [What is GDI+?](#)
 - [What about our image?](#)
- [Beginning of the Game.Draw method](#)
 - [GameTime information](#)
 - [The call to the GraphicsDevice.Clear method](#)
 - [The Color class](#)
 - [Type ARGB](#)
 - [Constructors, methods, and constants](#)

- [Code to draw the sprite](#)
 - [Honor the alpha values](#)
 - [Ignore the alpha values](#)
 - [Drawing the sprite\(s\)](#)
- [Overloaded Draw methods](#)
- [The SpriteBatch.End method](#)
- [Call Game.Draw on the superclass](#)
 - [Execute both versions of the overridden method](#)
 - [A required statement](#)
- [The end of the program](#)
- [Run the program](#)
- [Run my program](#)
- [Summary](#)
- [Miscellaneous](#)
- [Complete program listing](#)

Preface

This module is one in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

The modules were originally published for use with XNA 3.1 and have been upgraded for XNA 4.0. This upgrade had very little impact on earlier modules in this collection. However, beginning with this module, we begin to see major differences between version 3.1 and version 4.0 of XNA. In May of 2016, the modules are being updated for use with version **4.0 Refresh**. This will have very little impact on the modules. Here is what Microsoft has to say about the newer product:

Note: *Microsoft XNA Game Studio 4.0 Refresh updates XNA Game Studio 4.0 to fix bugs and add support for developing games that target Windows Phone OS 7.1 and developing games in Visual Basic.*

This course doesn't address Visual Basic or Windows Phone, but bug fixes are always welcome.

An earlier module titled [Getting Started](#) provided information on how to get started programming with Microsoft's XNA Game Studio.

Moving from C# to XNA

All of the modules prior to this one have been preparing you for this module. This is the module where we will begin applying what you have learned about C# and OOP to the XNA framework.

My objective

My objective is that you fully understand what you are doing when you write C# programs using the XNA framework. I don't want you to simply be filling in the blanks and hacking something together in hopes that you can find a combination of statements that seems to work. If we can accomplish that, you will be prepared to go much further into the sophisticated use of the XNA framework on your own after you complete the course.

Fasten your seatbelt

This module may be a rough ride from a technical viewpoint, so fasten your seatbelt, arm yourself with a pot of coffee, and let's go.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Initial game window.
- [Figure 2](#). Raw image with an (almost) transparent background.
- [Figure 3](#). Cropped upper-left corner of the game window
- [Figure 4](#). Cropped upper-left corner of the game window without honoring alpha transparency.
- [Figure 5](#). Select New Project on the Visual C# File menu.
- [Figure 6](#). Select a Windows Game project.
- [Figure 7](#). Solutions explorer and properties window exposed.
- [Figure 8](#). The Load method of the ContentManager class.

Listings

- [Listing 1](#). Initial contents of the file named Program.cs for a Console Application.
- [Listing 2](#). The file named Program.cs for a Windows Game project.
- [Listing 3](#). Initial contents of the file named Game1.cs.
- [Listing 4](#). Beginning of the class named Game1.
- [Listing 5](#). Constructor for the Game1 class.
- [Listing 6](#). The overridden LoadContent method.
- [Listing 7](#). Beginning of the Game.Draw method.
- [Listing 8](#). Draw the sprite.
- [Listing 9](#). Call Game.Draw on the superclass.
- [Listing 10](#). The Game1 class for the project named XNA0118Proj.

General background information

A software framework

XNA is a very sophisticated C# application. It isn't simply a program in the sense of the programs that you have seen so far in this course or a word processor or a spread sheet. Instead, it is a software **framework** designed specifically to make it easier for you to create computer games using the C# programming language.

What is a software framework?

Here is part of what [Wikipedia](#) has to say about a software framework:

Note: A software framework, in computer programming, is an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code providing specific functionality. Frameworks are a special case of software libraries in that they are reusable abstractions of code wrapped in a well-defined API, yet they contain some key distinguishing features that separate them from normal libraries.

Software frameworks have these distinguishing features that separate them from libraries or normal user applications:

1. **inversion of control** - In a framework, unlike in libraries or normal user applications, the overall program's flow of control is not dictated by the caller, but by the framework.
2. **default behavior** - A framework has a default behavior. This default behavior must actually be some useful behavior and not a series of no-ops.
3. **extensibility** - A framework can be extended by the user usually by selective overriding or specialized by user code providing specific functionality

4. **non-modifiable framework code** - The framework code, in general, is not allowed to be modified. Users can extend the framework, but not modify its code.

In short, a framework is *a computer program that helps you to write computer programs* . The description given above is a good match for the XNA framework.

New project choices in Visual C#

Up to this point in this collection of modules, all of the programs that I have explained have been *Visual C# Console Applications* .

Assuming that you have Visual C# 2010 and XNA Game Studio 4.0 Refresh installed on your computer, if you start Visual C# and select **New Project** from the **File** menu, you have a choice of about a dozen different kinds of projects that you can create within Visual C#. One of those choices is to create a **Console Application** .

A Console Application

When you create a Console Application, a project tree is created on your disk containing numerous folders and files. One of those files is a C# source code file named **Program.cs** . This file, which is the skeleton of a new class named **Program** , is opened in the editor pane in Visual C#.

The skeleton code in the file looks something like [Listing 1](#) when it first opens.

Note:

Listing 1 . Initial contents of the file named Program.cs for a Console Application.


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Start debugging

If you select **Start Debugging** on the **Debug** menu, the program will run and terminate almost immediately. At that point, the project tree will have expanded to contain more files and folders even though the program hasn't yet done anything useful.

The Main method

As you have learned in earlier modules, every C# program must have a **Main** method. Program execution begins and ends in the **Main** method. When control is in the **Main** method and it no longer has any code to execute, the program terminates. The **Main** method in [Listing 1](#) is empty, which explains why the program runs and terminates almost immediately.

The starting point for a Console Application

This is the starting point for creating a Console Application in Visual C#. To cause your program to exhibit some useful behavior, you may add code inside the **Main** method, add new methods to the **Program** class, define new classes, instantiate objects, call methods on those objects etc.

If you have studied the earlier modules in this collection, this will not be new information for you.

A Windows Game Application

Another choice that you have when creating a new project is a **Windows Game (4.0)** application.

Once again, a project tree is created on your disk but it contains more folders and files than the tree that is created for a console application. (Some of the files are initially empty.)

Source code files

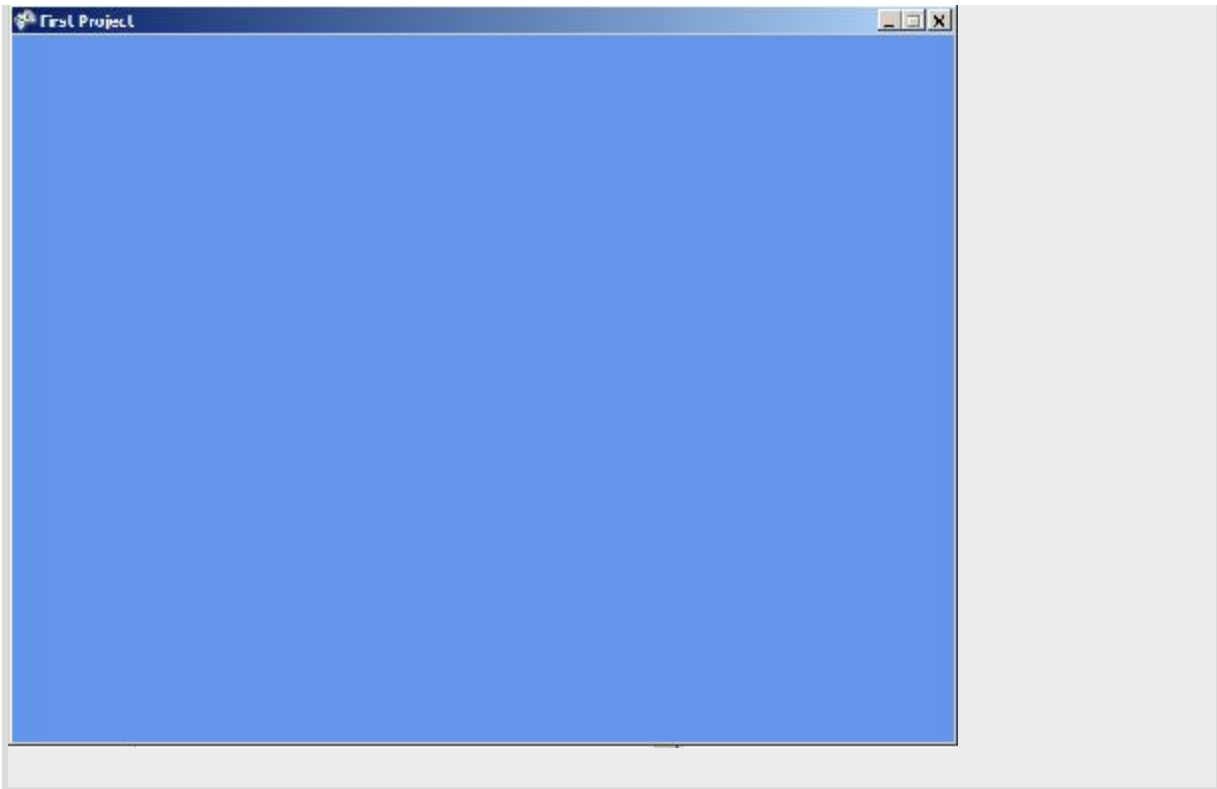
As before, there is a C# source code file named **Program.cs** along with another C# source code file named **Game1.cs**. Unlike before, however, the file named **Program.cs** is not opened in the editor pane of Visual C#. Instead, the file named **Game1.cs** is opened in the editor pane.

Build and debug

If you debug this program at this point, it does not run and terminate immediately like a console application. Instead it runs and displays a game window like the one shown in [Figure 1](#) (except that it is quite a bit larger).

Note:

Figure 1 . Initial game window.



The game window will remain on your computer screen until you terminate the program by clicking the X in the upper-right corner or terminate it in some other way.

The file named **Program.cs**

The file named **Program.cs** doesn't automatically open in the editor for one simple reason. The creators of XNA didn't intend for us to modify it. However, it will be instructive for us to take a look at it anyway. The source code contained in the file named **Program.cs** is shown in [Listing 2](#).

Note:

Listing 2 . The file named Program.cs for a Windows Game project.

```

using System;

namespace WindowsGame2
{
#if WINDOWS || XBOX
    static class Program
    {
        /// <summary>
        /// The main entry point for the
application.
        /// </summary>
        static void Main(string[] args)
        {
            using (Game1 game = new Game1())
            {
                game.Run();
            }
        }
    }
#endif
}

```

Instantiate a **Game1** object and call the **Run** method

The code in [Listing 2](#):

- Instantiates a new object of a class named **Game1** .
- Saves the object's reference in a reference variable named **game** .
- Calls the **Run** method on the reference to the **Game1** object

This code is inside the **Main** method in the file named **Program.cs** . The **Main** method runs when the program is started. The call to the **Run** method starts the game portion of the program running.

What do we know about the class named **Game1**?

We don't know anything about the class named **Game1** yet, but we will learn about it shortly. Before we get to that, however, I need to explain some unusual syntax in [Listing 2](#).

The using directive

You learned earlier that one of the benefits of the **using** directive is to eliminate the requirement to always type the namespace (such as **System**) when referring to a class that belongs to that namespace. That is the purpose of the **using** directive at the top of [Listing 2](#).

However, there is another benefit that derives from the **using** directive that may be more important.

Memory management

One of the big issues in game programming (or any kind of programming that makes use of graphics files, sound files, or other large resource files) is to make certain that the memory occupied by those resources is freed up as soon as the resource is no longer needed.

Without going into a lot of detail, the use of the **using** keyword inside the **Main** method in [Listing 2](#) will assure that the **Dispose** method is called to free up all of the memory occupied by the **Game1** object when control reaches the closing curly brace following the **using** keyword.

Don't modify the file named Program.cs

As an XNA game programmer, you shouldn't normally have any reason to modify the contents of the file named **Program.cs** . We need to leave it alone and modify the file named **Game1.cs** instead.

The Game1 class

I told you earlier that when you create a new **Windows Game** project, the file named **Game1.cs** is opened in the editor pane. [Listing 3](#) shows the contents of that file.

Note:

Listing 3 . Initial contents of the file named Game1.cs.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;

namespace WindowsGame2
{
    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class Game1 :
Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        public Game1()
        {
            graphics = new
GraphicsDeviceManager(this);
```

```

        Content.RootDirectory = "Content";
    }

    /// <summary>
    /// Allows the game to perform any
initialization
    /// it needs to before starting to run.
    /// This is where it can query for any
required
    /// services and load any non-graphic
    /// related content. Calling
base.Initialize
    /// will enumerate through any components
    /// and initialize them as well.
    /// </summary>
    protected override void Initialize()
    {
        // TODO: Add your initialization logic
here

        base.Initialize();
    }

    /// <summary>
    /// LoadContent will be called once per
game and is the place to load
    /// all of your content.
    /// </summary>
    protected override void LoadContent()
    {
        // Create a new SpriteBatch, which can
be used to draw textures.
        spriteBatch = new
SpriteBatch(GraphicsDevice);

        // TODO: use this.Content to load your
game content here
    }

```

```

    }

    /// <summary>
    /// UnloadContent will be called once per
game and is the place to unload
    /// all content.
    /// </summary>
    protected override void UnloadContent()
    {
        // TODO: Unload any non ContentManager
content here
    }

    /// <summary>
    /// Allows the game to run logic such as
updating the world,
    /// checking for collisions, gathering
input, and playing audio.
    /// </summary>
    /// <param name="gameTime">Provides a
snapshot of timing values.</param>
    protected override void Update(GameTime
gameTime)
    {
        // Allows the game to exit

if(GamePad.GetState(PlayerIndex.One).Buttons.Back=
=ButtonState.Pressed)
        this.Exit();

        // TODO: Add your update logic here

        base.Update(gameTime);
    }

    /// <summary>
    /// This is called when the game should

```



```

draw itself.
    /// </summary>
    /// <param name="gameTime">Provides a
snapshot of timing values.</param>
    protected override void Draw(GameTime
gameTime)
    {

GraphicsDevice.Clear(Color.CornflowerBlue);

        // TODO: Add your drawing code here

        base.Draw(gameTime);
    }
}
}

```

How to create a game

I will begin with a simplified explanation of how to create games using XNA.

Game1 extends Game

To begin with, note that the **Game1** class extends the **Game** class. We do not modify the **Game** class. Instead, we *modify the behavior* of the **Game** class by extending it and overriding some of its methods.

Main method calls the Run method

As you saw earlier in [Listing 2](#), the **Main** method in the file named **Program.cs** instantiates an object of the **Game1** class (shown in [Listing 3](#)) and calls the **Run** method on that object. That starts the program running.

The Run method

The class named **Game1** does not define and does not override a method named **Run** . However, it does inherit a method named **Run** from the **Game** class.

Therefore, when the **Run** method is called on the object of the **Game1** class, the version of the **Run** method that is defined in the superclass named **Game** is executed.

Override five methods

The skeleton code for the **Game1** class in [Listing 3](#) overrides the following five methods inherited from the **Game** class:

1. Initialize
2. LoadContent
3. UnloadContent
4. Update
5. Draw

Initialization

The first three methods contain code that is needed to get everything initialized at the start of the game play and to shut down the program at the end of the game play.

The game loop

Once the game is initialized, the **Run** method, or some method called by the **Run** method ping-pongs back and forth between calls to the overridden **Update** method and the overridden **Draw** method. (Note, however that the two methods don't necessarily take turns executing.)

Override Update for game logic

You override the **Update** method to create the program logic associated with game play. To accomplish this, you will likely need to define other methods, define other classes, instantiate objects of other classes, call methods on those objects, test the keyboard, test the mouse, etc. In other words, at this point *you need to know how to program in C#* .

Override the Draw method

You override the **Draw** method to cause the various graphics objects in your game to be rendered in the game window shown in [Figure 1](#).

This module includes an explanation of a very simple program that displays a green arrow sprite near the upper-left corner of the game window (see [Figure 3](#)).

Preview

I will create a simple Windows game application that imports the image shown in [Figure 2](#). Note that this is a rectangular image with an (almost) transparent background. (The values of the alpha bytes outside the blue elliptical shape are about 5.)

Note:

Figure 2 . Raw image with an (almost) transparent background.



If you would like to replicate my program using this image, you should be able to right-click on the image in [Figure 2](#), download it, and save it on your computer. You can save it under any name you choose but the file name extension should be **png** .

Program output

The program displays the image near the upper-left corner of the game window and honors the transparency of the background as shown in [Figure 3](#).

Note:

Figure 3 . Cropped upper-left corner of the game window.



What if you don't honor the transparent background?

[Figure 4](#) shows the result of causing the alpha transparency value to be ignored and allowing the pixels that are almost transparent in [Figure 3](#) to be opaque.

Note:

Figure 4 . Cropped upper-left corner of the game window without honoring alpha transparency.



Honoring alpha transparency is the default in XNA 4.0. [Figure 4](#) was created by setting the **Premultiply Alpha** property (see [Figure 7](#)) of the image named **gorightarrow.png** to a value of *False* and then re-running the program.

Not a very exciting program

This program isn't very exciting because there is no motion and no sound. The program simply draws the same image in the same position during every iteration of the game loop. Despite that, this program will give us the opportunity to drill down into the technical aspects of several areas of the XNA framework.

Discussion and sample code

Creating a new Windows Game project

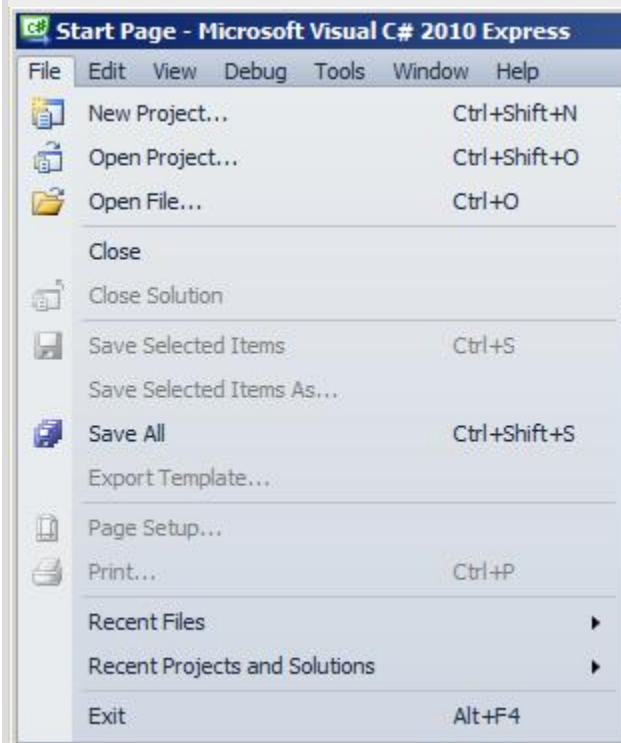
Before getting into the details of the code, I'm going to walk you through the steps involved in creating this Windows Game project using XNA.

Step 1: Create a new Windows Game project named XNA0118Proj

Pull down the Visual C# **File** menu and select **New Project** as shown in [Figure 5](#).

Note:

Figure 5 . Select New Project on the Visual C# File menu.

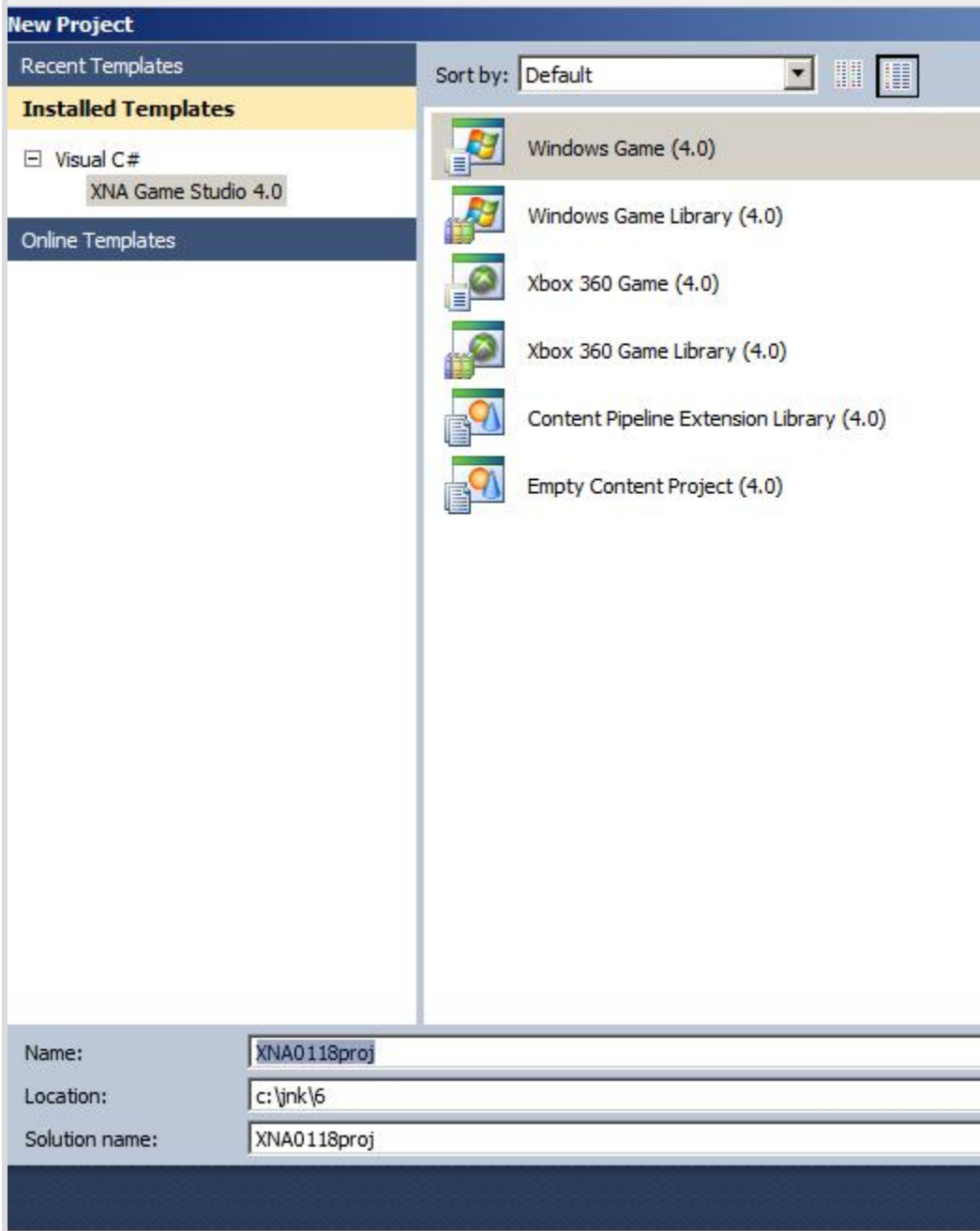


Select a Windows Game project

Select **XNA Game Studio 4.0** in the left pane of the **New Project** dialog. Select the **Windows Game (4.0)** icon and enter the name of your project in the **Name** field. Enter the storage location in the **Location** field and click the **OK** button that is off screen further to the right in [Figure 6](#).

Note:

Figure 6 . Select a Windows Game project.

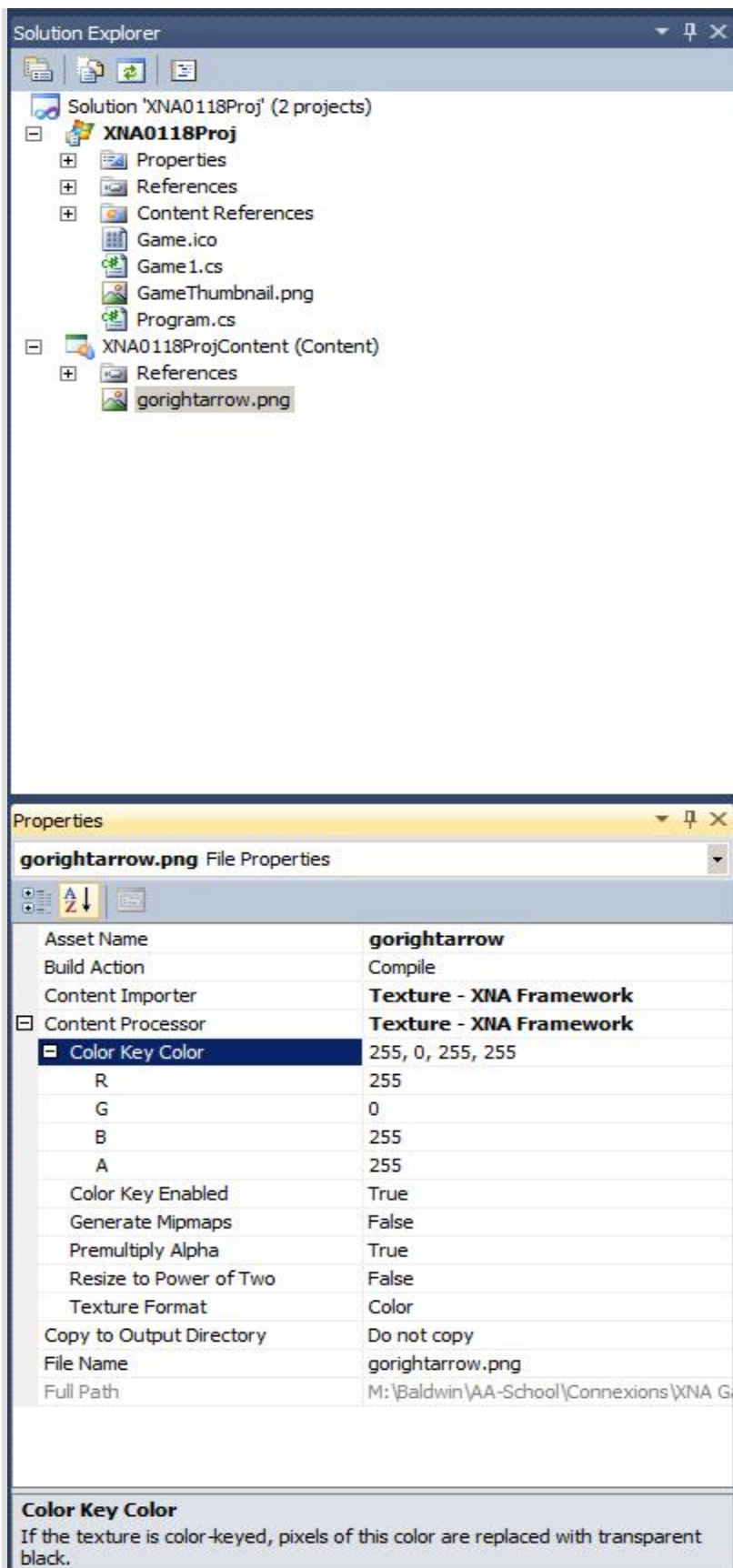


Step 2: Add your image file to the Content folder

If necessary, pull down the **View** menu and select **Other Windows** . Then select **Solution Explorer** and/or **Properties Window** so that they are exposed on the right side of the IDE as shown in [Figure 7](#) . (Note that the initial view of the **Properties** window is different from that shown in [Figure 7](#) .)

Note:

Figure 7 . Solutions explorer and properties window exposed.



Add your image to the Content folder

Assuming that your project is named **XNA0118proj** , right click on the **XNA0118projContent (Content)** folder in the **Solution Explorer** . Select **Add/Existing Item** in the dialog that follows. Browse to the image file that you are going to use and click the **Add** button. A copy of the image file should appear in the **Content** folder.

The Asset Name

Click the image file name in the **Content** folder and the information in the **Properties Window** should change to describe that file. Note the value of the **Asset Name** property in the **Properties Window** . You will need it later. (In this example, the value of the **Asset Name** in [Figure 7](#) is **gorightarrow** .)

Steps 3, 4, and 5: Write code

The next three steps involve writing code to upgrade the skeleton version of the class definition of the **Game1** class. I will explain that new code later. For now, the three steps for this example program are:

- Declare two instance variables named **myTexture** and **spritePosition** .
- Add a statement to the **LoadContent** method to load the image.
- Add statements to the **Draw** method to cause the image to be drawn in the game window.

Modify two overridden methods

As you saw in [Listing 3](#) , there are five overridden methods in the class definition for the **Game1** class that you can modify to customize the class for your game. This program modifies only two of those methods:

- **LoadContent**
- **Draw**

I will discuss those two methods along with some other material in this module. I will defer a detailed discussion of the other three methods until a future module when I write a program that modifies them.

Will discuss in fragments

A complete listing of the modified definition of the **Game1** class is provided in [Listing 10](#) near the end of the module. I will explain selected code fragments in the following paragraphs.

Beginning of the class named Game1

The beginning of the class definition for the class named **Game1** is shown in [Listing 4](#).

Note:

Listing 4 . Beginning of the class named Game1.

```
namespace XNA0118Proj{  
    public class Game1 :  
Microsoft.Xna.Framework.Game{
```

The namespace

This class definition belongs to the **XNA0118Proj** namespace. This is the name of the folder containing all of the other folders and files in the project tree as shown in the Solution Explorer in [Figure 7](#).

General information

The superclass named Game

The new class named **Game1** extends the existing class named **Game** . You will find the documentation for the class named **Game** [here](#) . The description of this class in the documentation is fairly modest. It says simply

Note: "Provides basic graphics device initialization, game logic, and rendering code."

Overridden methods

As I explained earlier, in order to write a program that runs under the XNA framework, you need to override some or all of five methods that are inherited into the **Game1** class from the **Game** class. Let's see some of what the documentation for the **Game** class has to say about these methods.

- **Initialize** - Called after the **Game** and **GraphicsDevice** are created, but before **LoadContent** .Override this method to query for any required services, and load any non-graphics resources. Use **LoadContent** to load graphics resources.
- **LoadContent** - Called when graphics resources need to be loaded. Override this method to load any game-specific graphics resources. This method is called by **Initialize** . Also, it is called any time the game content needs to be reloaded, such as when the **DeviceReset** event occurs.
- **UnloadContent** - Called when graphics resources need to be unloaded. Override this method to unload any game-specific graphics resources.
- **Update** - Called when the game has determined that game logic needs to be processed. This might include the management of the game state,

the processing of user input, or the updating of simulation data.
Override this method with game-specific logic.

- **Draw** - Called when the game determines it is time to draw a frame.
Override this method with game-specific rendering code.

The game loop

According to the documentation for the **Game** class,

" **Update** and **Draw** are called at different rates depending on whether **IsFixedTimeStep** is true or false.

If **IsFixedTimeStep** is false, **Update** and **Draw** will be called in a continuous loop.

If **IsFixedTimeStep** is true, **Update** will be called at the interval specified in **TargetElapsedTime** , while **Draw** will only be called if an **Update** is not due.

If **Draw** is not called, **IsRunningSlowly** will be set to true.

For more information on fixed-step and variable-step game loops, see [Application Model Overview](#)."

The big picture

Sifting through all of this detail in an attempt to get a big picture view, we see that we should:

- Override **Initialize** for any special initialization and for loading any non-graphic resources. For example, sound files are non-graphic resources.
- Override **LoadContent** to load all graphic resources.
- Override **UnloadContent** if any graphic resources need to be unloaded.

- Override **Update** to implement all of the game logic.
- Override **Draw** to draw an individual frame based on values created and stored by the overridden **Update** method, such as the current position of a sprite.
- Be aware that there are two different approaches to controlling the timing of the game loop, depending on whether the [IsFixedTimeStep](#) property of the **Game** object is true or false. The default value is true, meaning that the game will attempt to call the **Update** method on a fixed time interval even if that means that the **Draw** method doesn't get called during some iterations of the game loop.

The constructor for the Game1 class

[Listing 5](#) shows the declaration of two instance variables followed by the constructor for the **Game1** class. One of the instance variables is used in the constructor and the other is used later in the program.

Note:

Listing 5 . Constructor for the Game1 class.

```
GraphicsDeviceManager graphics;  
SpriteBatch spriteBatch;  
  
public Game1() {  
    graphics = new GraphicsDeviceManager(this);  
    Content.RootDirectory = "Content";  
} // end constructor
```

What is a constructor?

A constructor is a special method-like structure that is executed once and only once during the instantiation of an object.

The first statement in the **Main** method in [Listing 2](#) uses the **new** operator to cause the constructor to be executed. When the constructor completes its task, it returns a reference to the object just constructed. That reference is stored in the local reference variable of type **Game1** named **game** in [Listing 2](#).

A new GraphicsDeviceManager object

The first statement in the constructor in [Listing 5](#) instantiates a new object of the class [GraphicsDeviceManager](#) and stores that object's reference in the instance variable named **graphics** .

The documentation for [GraphicsDeviceManager](#) isn't very descriptive. Here is some of what Aaron Reed (the author of the *Learning XNA* books from O'Reilly) has to say on the topic.

Note: "This (GraphicsDeviceManager) is a very important object because it provides you, as a developer, with a way to access the graphics device on your ... The GraphicsDeviceManager object has a property called GraphicsDevice that represents the actual graphics device on your machine."

He goes on to explain how the **GraphicsDevice** object acts as a conduit between your XNA program and the physical graphics device on your machine.

Passing the **this** keyword as a parameter

Note the parameter that is passed to the **GraphicsDeviceManager** constructor in [Listing 5](#). The documentation tells us that the parameter

must be of type **Game** and is the **Game** that the **GraphicsDeviceManager** should be associated with.

I don't recall having discussed the keyword **this** earlier in this collection of modules. According to Jesse Liberty ([Programming C#](#) from O'Reilly)

Note: the keyword **this** is a variable that references the current instance of a class or struct.

Any time that the code in an instance method needs a reference to the object to which the method belongs, **this** is available as a reference to that object.

In this case, the code in [Listing 5](#) says to instantiate a new **GraphicsDeviceManager** object and to associate it with **this Game** object.

The ContentManager

A **Game** object has a property named **Content** that references an object of the class [ContentManager](#).

According to the documentation, the ContentManager

Note: "is the run-time component which loads managed objects from the binary files produced by the design time content pipeline. It also manages the lifespan of the loaded objects..."

The second statement in the constructor in [Listing 5](#) notifies the **ContentManager** that the folder named " **Content**" is the root of a directory tree in which content for the game will be stored. In this program,

we only have one item of content and it is the image file that was added to the **Content** folder earlier.

When the constructor in Listing 5 terminates...

When the constructor terminates, the new **Game1** object occupies memory and a reference to the object is stored in the variable named **game** in [Listing 2](#). The code in the **Main** method in [Listing 2](#) immediately calls the **Run** method on the **Game1** object's reference.

The **Game1** class neither defines nor overrides a method named **Run** . However, it does inherit a method named [Run](#) from the **Game** class. Therefore, the method named **Run** that is defined in the **Game** class is executed.

The Run method of the Game class

Here is what the documentation has to say about this method.

Note: "Call this method to initialize the game, begin running the game loop, and start processing events for the game."

This method calls the game **Initialize** and **BeginRun** methods before it begins the game loop and starts processing events for the game."

The BeginRun method

We already know about the **Initialize** method. Here is what the [documentation](#) has to say about the **BeginRun** method.

Note: "Called after all components are initialized but before the first update in the game loop."

The game loop

At this point, after the **Initialize** and the **LoadContent** methods have been called, either the **Run** method or the **BeginRun** method, or perhaps some other method that is called by one of those methods goes into a loop calling the **Update** method and the **Draw** method.

The timing and the order in which the two methods are called is determined by the value of **IsFixedTimeStep** as explained [earlier](#).

But, we're getting ahead of ourselves. We need to slow down and discuss the overridden **LoadContent** method.

The overridden LoadContent method

The overridden **LoadContent** method is shown in its entirety in [Listing 6](#) along with the declaration of two instance variables.

Note:

Listing 6 . The overridden LoadContent method.

```
//Declare two variables
Texture2D myTexture;
Vector2 spritePosition = new
Vector2(10.0f,15.0f);

protected override void LoadContent() {
    // Create a new SpriteBatch, which can be
used
    // to draw textures.
```

```
        spriteBatch = new
SpriteBatch(GraphicsDevice);
        //Load the image
        myTexture =
            Content.Load<Texture2D>
("gorightarrow");
    } //end LoadContent
```

Two new instance variables

[Listing 6](#) begins by declaring two new instance variables of types **Texture2D** and **Vector2** named **myTexture** and **spritePosition** respectively.

The variable named **myTexture** will be used in the **LoadContent** method of [Listing 6](#) to store a reference to a **Texture2D** object created from the image file with the **Asset Name** of **gorightarrow** (see [Figure 7](#)). It will also be used later in the overridden **Draw** method where it is the sprite being drawn.

The variable named **spritePosition** will be used later in the overridden **Draw** method to specify the location to draw the sprite. I will have more to say about this variable later.

The **Texture2D** class

Here is some of what the [documentation](#) has to say about the **Texture2D** class.

Note: "Represents a 2D grid of *texels* .

Note: A *texel* represents the smallest unit of a texture that can be read from or written to by the GPU (Graphics Processing Unit). A texel is composed of 1 to 4 components. Specifically, a texel may be any one of the available texture formats represented in the **SurfaceFormat** enumeration.

Note: A **Texture2D** resource contains a 2D grid of texels. Each texel is addressable by a u, v vector. Since it is a texture resource, it may contain mipmap levels."

You can view a diagram of a texture resource containing a single 3x5 texture with three mipmap levels [here](#). You can read what Wikipedia has to say about mipmaps [here](#). The image used in this program doesn't have any mipmaps.

The SurfaceFormat enumeration

The **SurfaceFormat** enumeration defines numeric values representing about 50 different types of surface formats such as **Rgba32**, which is defined as

Note: "(Unsigned format) 32-bit *RGBA* pixel format with alpha, using 8 bits per channel."

The Texture Format of my image

For example, the **Properties Window** in [Figure 7](#) shows the image that I used for my program to have a **Texture Format** property value of **Color**. The definition of the **SurfaceFormat** enumeration for **Color** is

Note: "(Unsigned format) 32-bit *ARGB* pixel format with alpha, using 8 bits per channel."

Note that this is similar to **Rgba32** except that the position of the alpha byte relative to the other three bytes is different.

A new **SpriteBatch** object

The code in the overridden **LoadContent** method of [Listing 6](#) begins by instantiating a new **SpriteBatch** object and saving its reference in the reference variable named **spriteBatch** . (That variable is declared at the top of [Listing 3](#).)

Note: Note that the statement that instantiates the **SpriteBatch** object in [Listing 6](#) is already in the skeleton of the **Game1** class when it first appears in the edit window of the Visual C# IDE. (See [Listing 3](#).)

The **SpriteBatch** class

According to the [documentation](#), an object of the **SpriteBatch** class

Note: "Enables a group of sprites to be drawn using the same settings."

The constructor for an object of the **SpriteBatch** class requires an incoming parameter that is a reference to the **graphicsDevice** of the current platform as type **GraphicsDevice** .

This can be confusing

GraphicsDevice is the name of an XNA class. It is also the name of a property of the **Game** class that is inherited into the **Game1** class. The parameter that is passed to the constructor for the **SpriteBatch** object in [Listing 6](#) is the inherited property.

The inherited property

The inherited property contains a reference to an object of the class **GraphicsDevice**, which is apparently populated in conjunction with the instantiation of the **GraphicsDeviceManager** object in the constructor of [Listing 5](#). However it gets populated, it is a reference to the **graphicsDevice** on the current platform. This causes the new **SpriteBatch** object to be aware of the **graphicsDevice** on the current platform. It will be used in the **Draw** method later to draw the sprite.

The new code

The last statement in [Listing 6](#) is the new code that I wrote into the overridden **LoadContent** method. The **Game1** class inherits a property of the **Game** class named **Content**. This property contains a reference to the current [ContentManager](#) object.

Therefore, the new code in [Listing 6](#) calls the **Load** method on the current **ContentManager** object.

Generic methods

Some methods in C# are known as [generic methods](#), and the [Load](#) method of the **ContentManager** class is one of them. The documentation describes the Load method as follows:

Note: "Loads an asset that has been processed by the Content Pipeline."

Required syntax for the Load method

[Figure 8](#) shows the syntax required for calling the **Load** method. This syntax was taken from the documentation.

Note:

Figure 8 . The Load method of the ContentManager class.

```
public virtual T Load<T> (string assetName)
```

What do the angle brackets mean?

To call this method, you must replace the T between the angle brackets in [Figure 8](#) with the type of asset to be loaded. According to the [documentation](#),

Note: " **Model** , **Effect** , **SpriteFont** , **Texture** , **Texture2D** , **Texture3D** and **TextureCube** are all supported by default by the standard Content Pipeline processor, but additional types may be loaded by extending the processor."

Calling the Load method of the current ContentManager

[Listing 6](#) calls the **Load** method, specifying an asset type of **Texture2D** , for the purpose of loading the content identified in [Figure 7](#) with an **Asset Name** property value of **gorightarrow** .

You will recall that this is the value given to the **Asset Name** property of the image file named **gorightarrow.png** when it was added to the Content folder earlier in this module.

Populate the variable named **myTexture**

The value returned from the **Load** method is assigned to the variable named **myTexture** in [Listing 6](#). It will be used later in the **Draw** method to draw the sprite in the game window as shown in [Figure 3](#) and [Figure 4](#).

That completes the definition of the overridden **LoadContent** method.

The **Vector2** structure

Returning to the variable declarations in [Listing 6](#), [Vector2](#) is a structure (similar to a class with no inheritance capability) containing two components of type **float** named **X** and **Y**.

In this program, the structure referred to by **spritePosition** in [Listing 6](#) is used to encapsulate the coordinates of the upper-left corner of the sprite (10.0f,15.0f) when the sprite is drawn in the game window as shown in [Figure 3](#) and more obviously in [Figure 4](#).

This variable will also be used later in the overridden **Draw** method.

The overridden **Game.Draw** method

That brings us to the **Draw** method inherited from the **Game** class, shown near the bottom of [Listing 3](#). According to the [documentation](#), this method is

Note: "Called when the game determines it is time to draw a frame. Override this method with game-specific rendering code."

Note that significant changes were made to the required contents of the **Game.Draw** method in XNA 4.0 as compared to XNA 3.1.

Game loop timing

As you learned earlier, **Update** and **Draw** are called at different rates depending on whether **IsFixedTimeStep** is true or false.

If **IsFixedTimeStep** is false, **Update** and **Draw** will be called sequentially as often as possible.

If **IsFixedTimeStep** is true, **Update** will be called at the interval specified in **TargetElapsedTime**, while **Draw** will continue to be called as often as possible. For more information on fixed-step and variable-step game loops, see [Application Model Overview](#)."

Because this program doesn't override the **Update** method, it doesn't matter how often the **Draw** method is called. Each time it is drawn, the sprite is drawn in the same position as shown in [Figure 3](#) and [Figure 4](#).

More general information

What is a sprite?

According to the [2D Graphics Overview](#),

Note: "Sprites are 2D bitmaps drawn directly on the screen, as opposed to being drawn in 3D space. Sprites are commonly used to display information such as health bars, number of lives, or text such as scores. Some games, especially older games, are composed entirely of sprites."

What is a bitmap?

According to the documentation for the [Bitmap](#) class,

Note: "A bitmap consists of the pixel data for a graphics image and its attributes. There are many standard formats for saving a bitmap to a file. GDI+ supports the following file formats: BMP, GIF, EXIF, JPG, PNG and TIFF."

What is GDI+?

According to the [documentation](#),

Note: "Microsoft Windows GDI+ is a class-based API for C/C++ programmers. It enables applications to use graphics and formatted text on both the video display and the printer. Applications based on the Microsoft Win32 API do not access graphics hardware directly. Instead, GDI+ interacts with device drivers on behalf of applications."

What about our image?

Working backwards through the above information, we started with an image file named **gorightarrow.png** . We manually added the file to the

Content folder producing a game asset with an **Asset Name** of **gorightarrow** (see [Figure 7](#)).

Then we called the **Load** method in [Listing 6](#) to load the contents of the file into an object of type **Texture2D** and saved that object's reference in the instance variable named **myTexture** . At that point in the process, we had converted the contents of our image file into a format that can be thought of as a sprite. The variable named **myTexture** contains a reference to our sprite.

Beginning of the **Game.Draw** method

The **Game.Draw** method begins in [Listing 7](#). I am referring to the method here as **Game.Draw** to distinguish it from the method named **SpriteBatch.Draw** , which we will encounter shortly.

Note:

Listing 7 . Beginning of the **Game.Draw** method.

```
protected override void Draw(GameTime
gameTime) {
    GraphicsDevice.Clear(Color.CornflowerBlue);
```

GameTime information

Each time the **Game.Draw** method is executed, the incoming parameter contains time information encapsulated in an object of type [GameTime](#) . According to the documentation, the **GameTime** object provides a

Note: "Snapshot of the game timing state expressed in values that can be used by variable-step (real time) or fixed-step (game time) games."

We won't be using this information in this module, so I won't pursue it further here. However, we will need the information in future modules when we write code to cause a sprite to be moved and/or animated.

The call to the `GraphicsDevice.Clear` method

The call to the **`GraphicsDevice.Clear`** method in [Listing 7](#) is contained in the skeleton code for the **`Game1`** class as shown in [Listing 3](#).

The **`GraphicsDevice`** class provides overloaded versions of the [Clear](#) method. According to the documentation, the version shown in [Listing 7](#)

Note: "Clears the viewport to a specified color."

This version of the **`Clear`** method requires a single incoming parameter of type **`Color`** .

The **`Color`** class

The documentation describes an object of the [Color](#) class as follows:

Note: "A **`Color`** object stores a 32-bit value that represents a color. The color value contains four, 8-bit components: alpha, red, green, and blue. The first 8 bits (the most significant) contain the alpha component, the next 8 bits contain the red component, the next 8 bits contain the green

component, and the next 8 bits (the least significant) contain the blue component. The 32-bit value is stored in a variable of type ARGB."

Type ARGB

We learned about the ARGB texture format [earlier](#). Although ARGB is referred to as a type in the above quotation, it is not a class. Rather, it is a type established using a C-style [typedef](#).

Constructors, methods, and constants

The **Color** class provides several overloaded constructors and numerous methods that allow you to perform various operations on a **Color** object.

One of the constructors allows you to create a **Color** object that represents the color of your choice by specifying the individual values of the alpha, red, green, and blue color components.

In addition, the class provides many constants that represent different colors, one of which is named **CornflowerBlue**. This is the background color of the game window shown in [Figure 1](#).

You can create **Color** objects representing those colors simply by calling out the name of the class and the name of the color as shown by the code in [Listing 7](#).

Code to draw the sprite

Three statements are required to draw one sprite and twelve statements are required to draw ten sprites with the same settings. The sequence consists of a **Begin** statement, one or more **SpriteBatch.Draw** statements, and one **End** statement.

[Listing 8](#) shows the code that is used to draw our sprite once each time the **Game.Draw** method is called. Note that the **SpriteBatch.Draw** method is called inside the **Game.Draw** method.

Note:

Listing 8 . Draw the sprite.

```
spriteBatch.Begin();  
spriteBatch.Draw(  
  
myTexture, spritePosition, Color.White);  
spriteBatch.End();
```

The image that we used to create the sprite is shown in raw form in [Figure 2](#) . This is a rectangular image with the pixels outside the blue area having an alpha value of about 5.

Honor the alpha values

As mentioned earlier, the default case is to honor the alpha values in XNA 4.0. This produces the output image shown in [Figure 3](#) .

Setting the **Premultiply Alpha** property value to False in [Figure 7](#) will cause the alpha value to be ignored. This will produce the output image shown in [Figure 4](#) .

Ignore the alpha values

As explained earlier, honoring alpha transparency is the default case in XNA 4.0. [Figure 4](#) was created by setting the **Premultiply Alpha** property (see [Figure 7](#)) of the image named **gorightarrow.png** to a value of *False*

and then re-running the program. This causes even the pixels with the very low alpha values to be opaque as shown in [Figure 4](#).

Drawing the sprite(s)

You can draw as many sprites as you need following the call to the **Begin** method in [Listing 8](#).

Each sprite drawn will be drawn according to the parameters passed to the **Begin** method.

If you need to draw some sprites with different parameters, call the **SpriteBatch.End** method and start the sequence over with a new call to the **SpriteBatch.Begin** method and new parameters.

In this case we only have one sprite to draw. [Listing 8](#) calls the **SpriteBatch.Draw** method to draw that sprite and then calls the **SpriteBatch.End** method to end the drawing sequence.

Overloaded Draw methods

There are several overloaded versions of the **SpriteBatch.Draw** method. According to the [documentation](#), the version used in [Listing 8](#)

Note: "Adds a sprite to the batch of sprites to be rendered, specifying the texture, screen position, and color tint. Before any calls to **Draw**, you must call **Begin**. Once all calls to **Draw** are complete, call **End**."

The code in [Listing 8](#) passes three parameters to the Draw method:

- **myTexture** - The sprite texture. (See [Listing 6](#).)

- **spritePosition** - The location, in screen coordinates, where the sprite will be drawn. (See [Listing 6](#).)
- **Color.White** - The color channel modulation to use. (Use Color.White for full color with no tinting.)

The SpriteBatch.End method

According to the [documentation](#), this method

Note: "Flushes the sprite batch and restores the device state to how it was before Begin was called. Call End after all calls to Draw are complete."

Call Game.Draw on the superclass

When you instantiate an object from a class that extends another class and overrides a method from the superclass, the new object contains both the original version and the overridden version of the method.

Execute both versions of the overridden method

Often it is desirable or necessary to cause both versions to be executed. The code in [Listing 9](#) shows the syntax used to cause an overridden method to call the original version of the method using the keyword **base**. The keyword **base** is a reference to that portion of the object that represents the properties, events, and methods of the superclass.

Note:

Listing 9 . Call Game.Draw on the superclass.


```
        base.Draw(gameTime);  
    }//end Draw method  
}//End class  
}//End namespace
```

A required statement

The statement shown in [Listing 9](#) is already contained in the skeleton code produced by Visual C# (see [Listing 3](#)).

The [documentation](#) for the **Game.Draw** method contains the following:

Note: "In classes that derive from **Game** , it is necessary to make these calls:

Note: Call **base.Draw** in **Draw** to enumerate through any graphics components that have been added to **Components** . This method will automatically call the **Initialize** method for every component that has been added to the collection."

We won't worry about the reason why we must do this at this point. We will simply follow the instructions and make the call.

The end of the program

That completes the explanation for this program. Because of the simplicity of the program, we had no need to override the following methods (see

[Listing 3](#)):

- Initialize
- UnloadContent
- Update

We will develop more complicated programs in future modules and will have a need to override one or more of these methods. I will explain them at that time.

Run the program

I encourage you to copy the code from [Listing 10](#). Use that code to create an XNA project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Run my program

Click [here](#) to download a zip file containing my version of the program. Extract the folder named **XNA0118Proj** from the zip file and save it somewhere on your disk. Start **Visual C# 2010 Express** and select **Open Project...** from the **File** menu. Navigate to the project folder and select the file with the extension of **.sln**. This should cause the project to open and be ready to run or debug as described in the earlier module titled [Getting Started](#).

Summary

In this module, I used a very simple XNA program to teach you many of the details regarding the incorporation of the XNA framework into the object-oriented C# programming language. I also taught you about constructors, the **this** keyword, the **base** keyword, and some of the differences between a Console Application and a Windows Game application.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Xna0118-The XNA Framework and the Game Class
- File: Xna0118.htm
- Published: 02/28/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the XNA program discussed in this module is provided in [Listing 10](#).

Note:

Listing 10 . The Game1 class for the project named XNA0118Proj.

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;

namespace XNA0118Proj{
    public class Game1 :
Microsoft.Xna.Framework.Game{
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        public Game1()
        {
            graphics = new
GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

        protected override void Initialize()
        {
            // TODO: Add your initialization logic
here
```

```

        base.Initialize();
    }

    //Declare two variables
    Texture2D myTexture;
    Vector2 spritePosition = new
Vector2(10.0f, 15.0f);

    protected override void LoadContent()
    {
        // Create a new SpriteBatch, which can
be used
        // to draw textures.
        spriteBatch = new
SpriteBatch(GraphicsDevice);
        //Load the image
        myTexture =
            Content.Load<Texture2D>
("gorightarrow");
    } //end LoadContent

    protected override void UnloadContent()
    {
        // TODO: Unload any non ContentManager
content here
    }

    protected override void Update(GameTime
gameTime)
    {
        // Allows the game to exit

        if(GamePad.GetState(PlayerIndex.One).Buttons.Back==
ButtonState.Pressed)
            this.Exit();

        // TODO: Add your update logic here

```

```
        base.Update(gameTime);
    }

    protected override void Draw(GameTime
gameTime)
    {
GraphicsDevice.Clear(Color.CornflowerBlue);

        spriteBatch.Begin();
        spriteBatch.Draw(myTexture,
spritePosition, Color.White);
        spriteBatch.End();

        base.Draw(gameTime);
    } //end Draw method
} //End class
} //End namespace
```

-end-

Xna0120-Moving Your Sprite and using the Debug Class
Learn how to move your sprite. Also learn how to use methods of the Debug class.

Revised: Sun May 08 09:19:08 CDT 2016

This page is part of a Book titled [XNA Game Studio](#).

Table of Contents

- [Table of Contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [Skeleton code](#)
 - [Override some or all of five methods](#)
- [Preview](#)
 - [Moving and bouncing sprite](#)
 - [The Debug class](#)
- [Discussion and sample code](#)
 - [Making the sprite move](#)
 - [The game loop](#)
 - [Distance to move](#)
 - [How often is the Update method called?](#)
 - [If the IsFixedTimeStep property is false...](#)
 - [If the IsFixedTimeStep property is true...](#)

- [If the Draw method is not called...](#)
- [The overridden Update method](#)
 - [Xbox 360 code](#)
 - [Test for sprite out of bounds horizontally](#)
 - [Test for a collision of the sprite with an edge](#)
 - [Test for sprite out of bounds vertically](#)
 - [Change the current position of the sprite](#)
 - [Move to the right and down](#)
 - [Reverse the direction of motion](#)
 - [No visible movement at this time](#)
 - [The remainder of the overridden Update method](#)
- [Using the Debug class](#)
 - [Namespace considerations](#)
 - [Overridden LoadContent method](#)
 - [Overloaded WriteLine methods](#)
 - [Execute once only](#)
 - [The output](#)
- [Run the program](#)
- [Run my_program](#)
- [Summary](#)
- [Miscellaneous](#)
- [Complete program listing](#)

Preface

This module is one in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

An earlier module titled [Getting Started](#) provided information on how to get started programming with Microsoft's XNA Game Studio.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Moving and bouncing sprite.
- [Figure 2](#). Debug output window.

Listings

- [Listing 1](#). Skeleton code for a new Windows Game project.
- [Listing 2](#). Declare variables that specify the incremental distance to move.
- [Listing 3](#). Beginning of the overridden Update method.
- [Listing 4](#). Test for sprite out of bounds horizontally.
- [Listing 5](#). Test for sprite out of bounds vertically.
- [Listing 6](#). Change the current position of the sprite.
- [Listing 7](#). The remainder of the overridden Update method.
- [Listing 8](#). Overridden LoadContent method.
- [Listing 9](#). The class named Game1 for the project named XNA0120Proj.

General background information

The earlier module titled [Xna0118-The XNA Framework and the Game Class](#) used a very simple XNA program to teach many of the details regarding the incorporation of the XNA framework into the object-oriented C# programming language.

Skeleton code

When you create a new **Windows Game** project using Visual C#, a source code file named **Game1.cs** is automatically created and opened in the editor window. The file contains skeleton code for a windows game based on XNA. [Listing 1](#) shows the skeleton code contained in the file named **Game1** (with most of the comments deleted for brevity).

Note:

Listing 1 . Skeleton code for a new Windows Game project.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;

namespace WindowsGame2
{
    public class Game1 :
Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        public Game1()
        {
            graphics = new
GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }
    }
}
```

```

        protected override void Initialize()
        {
            // TODO: Add your initialization logic
here

            base.Initialize();
        }

        protected override void LoadContent()
        {
            // Create a new SpriteBatch, which can
be used to draw textures.
            spriteBatch = new
SpriteBatch(GraphicsDevice);

            // TODO: use this.Content to load your
game content here
        }

        protected override void UnloadContent()
        {
            // TODO: Unload any non ContentManager
content here
        }

        protected override void Update(GameTime
gameTime)
        {
            // Allows the game to exit

            if(GamePad.GetState(PlayerIndex.One).Buttons.Back==
ButtonState.Pressed)
                this.Exit();

            // TODO: Add your update logic here

            base.Update(gameTime);

```

```

    }

    protected override void Draw(GameTime
gameTime)
    {
GraphicsDevice.Clear(Color.CornflowerBlue);

        // TODO: Add your drawing code here

        base.Draw(gameTime);
    }
}

```

Override some or all of five methods

To create a new game program, you override some or all of the five methods shown in [Listing 1](#).

The program that I explained in the earlier module overrode the **LoadContent** and **Draw** methods in such a way as to create a sprite and display it in the upper-left corner of the game window. However, that program did not override the **Update** method. The sprite did not move and was not animated.

Preview

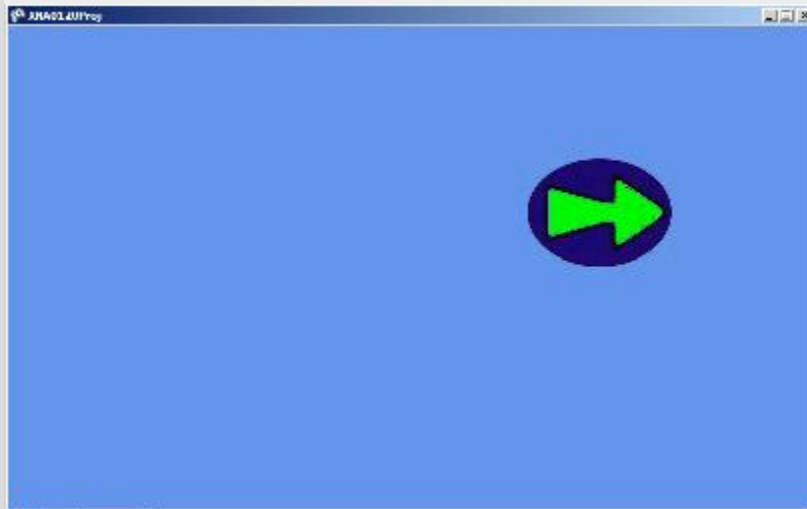
Moving and bouncing sprite

The **Update** method is used to implement game logic. The **Draw** method is used to render the current state of the game on the computer screen. In this module, I will override the **Update** method to cause the sprite to move around the game window and to bounce off the edges of the game window.

[Figure 1](#) shows a reduced screen shot of the sprite moving in the game window.

Note:

Figure 1 . Moving and bouncing sprite.



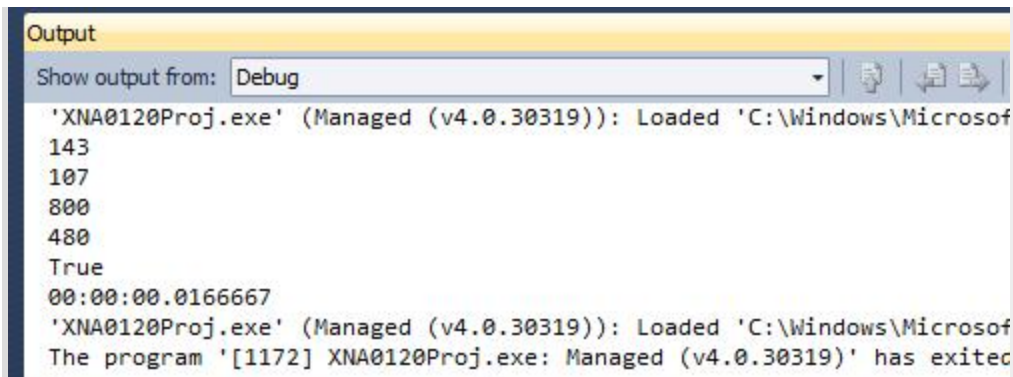
The Debug class

On a completely unrelated note, I will introduce you to the **Debug** class and show you how to use the **WriteLine** method of the **Debug** class to display information while the program is running.

[Figure 2](#) shows a screen shot of the **Debug** output window in the lower left corner of the Visual C# IDE. I will explain the values that you see in [Figure 2](#) later in this module.

Note:

Figure 2 . Debug output window.



```
Output
Show output from: Debug
'XNA0120Proj.exe' (Managed (v4.0.30319)): Loaded 'C:\Windows\Microsof
143
107
800
480
True
00:00:00.0166667
'XNA0120Proj.exe' (Managed (v4.0.30319)): Loaded 'C:\Windows\Microsof
The program '[1172] XNA0120Proj.exe: Managed (v4.0.30319)' has exited
```

Discussion and sample code

I will explain the code in fragments, and I will only explain those fragments that are different from the code that I explained in the earlier module titled [Xna0118-The XNA Framework and the Game Class](#). A complete listing of the code for the class named **Game1** is provided in [Listing 9](#) near the end of the module.

Making the sprite move

Sprite motion in an XNA game is accomplished by changing the current position coordinates of the sprite in the **Update** method and drawing the sprite in its new position in the **Draw** method.

The game loop

You learned about the XNA game loop in the earlier module .

In order to make the sprite move, we need to override the **Update** method to cause the sprite's position coordinates to change during each iteration of the game loop. The code to accomplish this begins in [Listing 2](#).

Note:

Listing 2 . Declare variables that specify the incremental distance to move.

```
//Specify the distance in pixels that the  
sprite  
// will move during each iteration.  
int stepsX = 5;  
int stepsY = 3;
```

Distance to move

[Listing 2](#) declares and populates two instance variables that specify the incremental distance that the sprite will move each time the **Update** method is called. The horizontal and vertical incremental distances are 5 pixels and 3 pixels respectively.

Note: Note that these two variables are physically declared between the **UnloadContent** and **Update** methods. Because they are declared inside a class but outside of a method or constructor, they are *instance* variables.

How often is the Update method called?

In this program, the computer will do its best to cause the **Update** method to be called once every 16.67 milliseconds or 60 times per second.

The **Update** and **Draw** methods are called at different rates depending on whether the **Game** property named **IsFixedTimeStep** is true or false.

If the **IsFixedTimeStep** property is false...

If the **IsFixedTimeStep** property is false, the **Update** and **Draw** methods will be called in a continuous loop. The repetition rate of the loop will depend on how long it takes the code in the loop to execute. That is not the case in this program because the value of the **IsFixedTimeStep** property is true, which is the default value.

If the **IsFixedTimeStep** property is true...

If the **IsFixedTimeStep** property is true, the **Update** method will be called at the interval specified in the property named **TargetElapsedTime** , while the **Draw** method will only be called if an **Update** is not due.

In this program, the value of the **TargetElapsedTime** property is 0.0166667 seconds, which is the default value.

If the **Draw** method is not called...

If the **Draw** method is not called (meaning that the computer can't keep up with the demands of the **Update** method), the property named **IsRunningSlowly** will be set to true. This property can be tested by the program during development to expose potential timing problems in the game loop.

The overridden Update method

The overridden Update method begins in [Listing 3](#).

Note:

Listing 3 . Beginning of the overridden Update method.

```
protected override void Update(GameTime gameTime) {
```



```
// Allows the game to exit

if(GamePad.GetState(PlayerIndex.One).Buttons.Back ==
ButtonState.Pressed)
    this.Exit();
```

If you compare [Listing 3](#) with [Listing 1](#), you will see that the code in [Listing 3](#) is contained in the skeleton code for the **Game1** class that is generated by Visual C#.

Xbox 360 code

The code in [Listing 3](#), which references a method of the [GamePad](#) class "Allows retrieval of user interaction with an Xbox 360 Controller." Since we are working on a PC game, we aren't really interested in the code in the body of the **Update** method in [Listing 3](#). The new code in our program begins in [Listing 4](#).

Test for sprite out of bounds horizontally

The code in [Listing 4](#) tests to determine if the sprite has encountered the right or left sides of the game window. If so, the sign is changed on **stepsX** to cause the sprite to reverse its horizontal direction of motion.

Note:

Listing 4 . Test for sprite out of bounds horizontally.

```
if(((spritePosition.X + myTexture.Width) >
Window.ClientBounds.Width) ||
```

```
(spritePosition.X  
< 0)){  
    stepsX *= -1;//Out of bounds, reverse  
direction  
    }//end if
```

The value of **Window.ClientBounds.Width** is the width of the game window in pixels.

The value of **spritePosition.X** specifies the current horizontal position of the upper-left corner of the sprite relative to the upper-left corner of the game window.

Note: Even though the sprite appears to have an elliptical shape in [Figure 1](#), it actually has a rectangular shape with all of the pixels outside the blue elliptical shape being almost transparent. You learned about this in the earlier module titled Xna0118-The XNA Framework and the Game Class

The value of **myTexture.Width** is the width of the sprite. Therefore, the sum of **spritePosition.X** and **myTexture.Width** specifies the current position of the upper-right corner of the sprite.

Test for a collision of the sprite with an edge

[Listing 4](#) tests to determine if the upper-right corner of the sprite has encountered the right edge of the game window or if the upper-left corner of the sprite has encountered the left edge of the game window. If so, the sign of **stepsX** is changed to cause the direction of motion of the sprite to be reversed. You will see how the change in sign causes a reversal in the direction of motion shortly.

Test for sprite out of bounds vertically

The code in [Listing 5](#) tests to determine if the sprite has encountered the bottom or top edges of the game window. If so, the sign is changed on **stepsY** to cause the sprite to reverse its vertical direction of motion.

Note:

Listing 5 . Test for sprite out of bounds vertically.

```
        if(((spritePosition.Y + myTexture.Height) >
Window.ClientBounds.Height) ||
        (spritePosition.Y
< 0)) {
        stepsY *= -1;//Out of bounds, reverse
direction
        }//end if
```

The logic in [Listing 5](#) is essentially the same as the logic in [Listing 4](#) with the difference being that [Listing 5](#) deals with vertical motion and the top and bottom edges of the game window instead of horizontal motion and the left and right edges of the game window.

Change the current position of the sprite

[Listing 6](#) changes the current position of the sprite by adding the incremental distances, **stepX** and **stepY** , to the current horizontal and vertical coordinates of the sprite.

Note:

Listing 6 . Change the current position of the sprite.

```
spritePosition.X += stepsX;//move horizontal  
spritePosition.Y += stepsY;//move vertical
```

Move to the right and down

The incremental distances were initialized with positive values in [Listing 2](#). Adding positive incremental distance values to the current coordinate values causes the position of the sprite to move to the right and down.

Reverse the direction of motion

The logic in [Listing 4](#) and [Listing 5](#) reverses the sign on the incremental distance values when the sprite encounters a left, right, top, or bottom edge of the game window.

Adding a negative incremental distance value to the current horizontal coordinate of the sprite causes it to be moved to the left. Adding a negative incremental distance value to the current vertical coordinate of the sprite causes it to be moved up the screen.

No visible movement at this time

Simply changing the current horizontal and vertical coordinates of the sprite does not produce a visible change in position. The visible change in position happens later in the **Draw** method when the game window is cleared to a constant color and the sprite is redrawn in the game window in its new position.

The **Draw** method in this program was not modified relative to the version in the earlier module titled [Xna0118-The XNA Framework and the Game Class](#). You can view the code in the **Draw** method in Listing 9 near the end of the module.

The remainder of the overridden Update method

The remainder of the overridden **Update** method is shown in [Listing 7](#).

Note:

Listing 7 . The remainder of the overridden Update method.

```
//The following statement is always  
required.  
    base.Update(gameTime);  
} //end Update
```

The statement in [Listing 7](#), which calls the **Update** method of the superclass, is always required in the overridden version. This code is placed in the overridden **Update** method in the skeleton code that is generated by Visual C# (see [Listing 1](#)) when you create a new Windows Game project.

Using the Debug class

One of the must useful debugging tools for relatively simple programs is the ability to get information displayed while the program is running. First I will introduce you to the Microsoft Support website titled [How to trace and debug in Visual C#](#). You will find a great deal of useful information there.

Next I will introduce you to the [Debug](#) class, which "Provides a set of methods and properties that help debug your code." There are some very useful tools there also.

Finally, I will show you some examples of using methods from the **Debug** class.

Namespace considerations

The **Debug** class is in the **System.Diagnostics** namespace. Therefore, you will either need to include that namespace in your "using" list or qualify every reference to the **Debug** class with the name of the namespace. I chose to include the namespace in my "using" list for this example.

Overridden LoadContent method

[Listing 8](#) shows my overridden **LoadContent** method. Only the code near the end of the method is different from the version of the **LoadContent** method that I explained in the earlier module titled [Xna0118-The XNA Framework and the Game Class](#).

Note:

Listing 8 . Overridden LoadContent method.

```
protected override void LoadContent() {
    //Create a new SpriteBatch, which can be
used to
    // draw textures.
    spriteBatch = new
SpriteBatch(GraphicsDevice);
    //Load the image
    myTexture = Content.Load<Texture2D>(
"gorightarrow");
    //Debug code for illustration purposes.
    Debug.WriteLine(myTexture.Width);
    Debug.WriteLine(myTexture.Height);
    Debug.WriteLine(Window.ClientBounds.Width);
    Debug.WriteLine(Window.ClientBounds.Height);
    Debug.WriteLine(IsFixedTimeStep);
    Debug.WriteLine(TargetElapsedTime);
} //end LoadContent
```

Overloaded WriteLine methods

The **Debug** class provides overloaded versions of the **WriteLine** method, which can be used to display information in the lower-left corner of the Visual C# IDE. These are static methods so they can be called simply by joining the name of the class to the name of the method as shown in [Listing 8](#).

Execute once only

I elected to put the code to illustrate this capability in the **LoadContent** method so that it would execute once and only once. Had I put it in either the **Update** method or the **Draw** method, it would have tried to execute during every iteration of the game loop which would not be satisfactory.

If you need to display information from inside the game loop, you will probably also need to use a counter with some logic to cause the information to be displayed only every nth iteration.

The output

The output from each of the six **WriteLine** statements in [Listing 8](#) (plus some other stuff) is shown in [Figure 2](#). As you can see, the sprite referred to by **myTexture** is 143 pixels wide and 107 pixels high. The game window is 800 pixels wide and 480 pixels high. As I mentioned earlier, the value of the **IsFixedTimeStep** property is True, and the value of the **TargetElapsedTime** property is 0.0166667 seconds.

Run the program

I encourage you to copy the code from [Listing 9](#). Use that code to create an XNA project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Run my program

Click [here](#) to download a zip file containing my version of the program. Extract the folder named **XNA0120Proj** from the zip file and save it somewhere on your disk. Start **Visual C# 2010 Express** and select **Open Project...** from the **File** menu. Navigate to the project folder and select the file with the extension of **.sln** . This should cause the project to open and be ready to run or debug as described in the earlier module titled [Getting Started](#).

Summary

In this module, you learned how to make it appear that a sprite is moving by changing the current position coordinates of the sprite in the **Update** method and drawing the sprite in the new position in the **Draw** method. This takes place once during each iteration of the game loop.

You also learned how to use the **WriteLine** method of the **Debug** class to display information while the program is running.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Xna0120-Moving Your Sprite and using the Debug Class
- File: Xna0120.htm
- Published: 02/28/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the XNA program discussed in this module is provided in [Listing 9](#).

Note:

Listing 9. The class named Game1 for the project named XNA0120Proj.

```
/*Project XNA0120Proj  
 * 12/27/09 R.G.Baldwin
```

```

*****
*****/

using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;
using System.Diagnostics;//to access Debug

namespace XNA0120Proj {

    public class Game1 :
Microsoft.Xna.Framework.Game {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        public Game1() {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }// end constructor

        protected override void Initialize() {
            //No initialization needed
            base.Initialize();
        }//end Initialize

        //Declare two variables
        Texture2D myTexture;
        Vector2 spritePosition = new

```

```

Vector2(10.0f,15.0f);

    protected override void LoadContent() {
        //Create a new SpriteBatch, which can be
used to
        // draw textures.
        spriteBatch = new
SpriteBatch(GraphicsDevice);
        //Load the image
        myTexture = Content.Load<Texture2D>(
"gorightarrow");

        //Debug code for illustration purposes.
        Debug.WriteLine(myTexture.Width);
        Debug.WriteLine(myTexture.Height);
        Debug.WriteLine(Window.ClientBounds.Width);
        Debug.WriteLine(Window.ClientBounds.Height);
        Debug.WriteLine(IsFixedTimeStep);
        Debug.WriteLine(TargetElapsedTime);
    } //end LoadContent

    protected override void UnloadContent() {
        //No unload code needed.
    } //end UnloadContent

    //Specify the distance in pixels that the
sprite
    // will move during each iteration.
    int stepsX = 5;
    int stepsY = 3;
    protected override void Update(GameTime
gameTime) {
        // Allows the game to exit

    if(GamePad.GetState(PlayerIndex.One).Buttons.Back
        ==

```

```

ButtonState.Pressed)
    this.Exit();
    //New code begins here.

    //Test to determine if the sprite moves out
of the
    // game window on the right or the left.
    if(((spritePosition.X + myTexture.Width) >
Window.ClientBounds.Width) ||
                                           (spritePosition.X
< 0)){
        stepsX *= -1;//Out of bounds, reverse
direction
    }//end if

    //Test to determine if the sprite moves out
of the
    // game window on the bottom or the top.
    if(((spritePosition.Y + myTexture.Height) >
Window.ClientBounds.Height) ||
                                           (spritePosition.Y
< 0)) {
        stepsY *= -1;//Out of bounds, reverse
direction
    }//end if
    spritePosition.X += stepsX;//move horizontal
    spritePosition.Y += stepsY;//move vertical

    //The following statement is always
required.
    base.Update(gameTime);
} //end Update

protected override void Draw(GameTime
gameTime) {

```

```
        GraphicsDevice.Clear(Color.CornflowerBlue);

        // Draw the sprite.
        spriteBatch.Begin();
        spriteBatch.Draw(
myTexture, spritePosition, Color.White);
        spriteBatch.End();

        //This statement is always required.
        base.Draw(gameTime);
    }//end Draw method
} //End class
} //End namespace
```

-end-

Xna0122-Frame Animation using a Sprite Sheet

In this module you will learn: How to create frame animation using a sprite sheet, how to flip and scale sprite images when they are drawn, how to implement different animation frame rates in the same program, how to work with different groups of sprite images in the same program, and how to change the size of the game window.

Revised: Sun May 08 14:07:44 CDT 2016

This page is part of a Book titled [XNA Game Studio](#).

Table of Contents

- [Table of Contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [A sprite sheet](#)
 - [Hundreds of sprite images](#)
 - [Frame animation](#)
 - [Downloading the sprite sheet](#)
- [Preview](#)
- [Discussion and sample code](#)
 - [The class named Game1](#)
 - [The modified constructor for the Game1 class](#)
 - [The overridden LoadContent method](#)
 - [Load the image](#)
 - [Initialization of variables](#)

- [spriteCol and spriteRow](#)
 - [frameWidth and frameHeight](#)
 - [msPerFrame](#)
 - [spriteEffect](#)
 - [winWidth](#)
- [The overridden Update method](#)
 - [The update method is fairly complex](#)
 - [The animation frame rate](#)
 - [Many drawings will be repeated](#)
 - [The gameTime parameter](#)
 - [The ElapsedGameTime property](#)
 - [The TimeSpan structure](#)
 - [Accumulate and compare elapsed time](#)
 - [Compute the location of the sprite to draw](#)
 - [The overall animation cycle](#)
 - [Two animation cycles from the bottom row of sprite images](#)
 - [Pause and animate in the same location](#)
 - [Some complex logic](#)
 - [Adjust column and row counters](#)
 - [Increment the column counter and compare](#)
 - [Execute the pause sequence if it is time for it](#)
 - [The conditional clause](#)
 - [Set the row counter to 1](#)
 - [Increment the pauseSequenceCnt](#)
 - [Adjust sprite position and frame rate](#)
 - [More complex logic](#)
 - [Scaling](#)
 - [The slide variable](#)

- [The sign of the variable named slide](#)
 - [Move the sprite image back and forth across the game window](#)
 - [Test for a collision with an edge](#)
 - [The spriteEffect variable](#)
- [The overridden Draw method](#)
 - [Begin the drawing process](#)
 - [Call the SpriteBatch.Draw method](#)
 - [The first two parameters](#)
 - [Three interesting parameters](#)
 - [The remaining four parameters](#)
 - [The rectangle](#)
 - [The upper left corner of the rectangle](#)
 - [The width and the height of the rectangle](#)
 - [The horizontal and vertical scale factors](#)
 - [A new object](#)
 - [Four times larger](#)
 - [Causing the sprite to face in the correct direction](#)
 - [Call the SpriteBatch.End method](#)
- [The visual frame rate](#)
 - [The fast frame rate](#)
 - [The slow frame rate](#)
 - [Avoid flicker but animate more slowly](#)
- [The end of the program](#)
- [Run the program](#)
- [Run my program](#)
- [Summary](#)

- [Miscellaneous](#)
- [Complete program listing](#)

Preface

This module is one in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

An earlier module titled [Getting Started](#) provided information on how to get started programming with Microsoft's XNA Game Studio.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Sprite sheet used to animate a dog.
- [Figure 2](#). A top row image.
- [Figure 3](#). A bottom row image.
- [Figure 4](#). A bottom row image flipped horizontally.
- [Figure 5](#). A top row image flipped horizontally.

Listings

- [Listing 1](#). Beginning of the class named Game1.
- [Listing 2](#). The constructor for the Game1 class.
- [Listing 3](#). The overridden LoadContent method.
- [Listing 4](#). Beginning of the Update method.

- [Listing 5](#). Compute the location of the sprite to draw.
- [Listing 6](#). Adjust column and row counters.
- [Listing 7](#). Execute the pause sequence if it is time for it.
- [Listing 8](#). Adjust sprite position and frame rate.
- [Listing 9](#). Move the sprite image back and forth across the game window.
- [Listing 10](#). Beginning of the overridden Draw method.
- [Listing 11](#). Begin the drawing process.
- [Listing 12](#). Call the SpriteBatch.Draw method.
- [Listing 13](#). The end of the program.
- [Listing 14](#). Class Game1 from the project named XNA0122Proj.

General background information

Frame animation typically involves displaying a series of images one at a time in quick succession where each image is similar to but different from the one before it. For example, [Figure 1](#) shows a series of images of a small dog running, jumping, stopping to answer nature's call, and then scratching the ground as dogs are prone to do.

Note:

Figure 1 . Sprite sheet used to animate a dog.



Note:

Credit for sprite artwork

I have no recollection of when and where I acquired this sprite sheet. If you are the artist that drew these sprites, please contact me and identify the original source and I will gladly give you credit for the artwork.

A sprite sheet

The format that you see in [Figure 1](#) is a small scale version of a format that is commonly known as a sprite sheet. If you Google "animation sprite sheet", you will find hundreds and possibly thousands of examples of animation sprite sheets on the web.

Hundreds of sprite images

Many of the sprite sheets that you will find on the web will contain hundreds of individual images usually arranged in several groups. One group may have several images that can be animated to create the illusion of a character running. Another group may have several images that can be animated to create the illusion of the character engaging in a martial arts battle. Other groups can be animated to create the illusion of other activities.

There are two groups of sprite images in [Figure 1](#). The images in the top row can be animated to show the dog running, jumping, playing and generally having fun.

The images in the bottom row can be animated to show the dog answering nature's call.

Frame animation

By displaying the individual images from a group sequentially with an appropriate time delay between images, you can create the illusion that the

character is engaging in some particular activity. When displayed in this manner, each image is often referred to as a frame. The overall process is often referred to as frame animation.

Downloading the sprite sheet

If you would like to replicate my program using the same sprite sheet, you should be able to right-click on [Figure 1](#) and save the image on your disk. Be sure to save it as an image file of type JPEG.

Preview

I will explain a program in this module that causes the dog to run back and forth across a small game window always facing in the correct direction as shown in [Figure 2](#). [Figure 2](#) through [Figure 5](#) show four random screen shots taken while the program was running.

Note:

Figure 2 . A top row image.



Note:

Figure 3 . A bottom row image.



Note:

Figure 4 . A bottom row image flipped horizontally.



Note:

Figure 5 . A top row image flipped horizontally.



You should be able to correlate the images of the dog shown in [Figure 2](#) through [Figure 5](#) with the individual images shown in [Figure 1](#). Note, however, that the images in [Figure 4](#) and [Figure 5](#) were flipped horizontally

so that the dog would be facing the correct way when moving from left to right across the game window.

Discussion and sample code

I will explain the code in this program in fragments, and I will only discuss the code that I modified relative to the skeleton code produced by Visual C# when I created the project. A complete listing of the file named **Game1.cs** is shown in [Listing 14](#) near the end of the module.

The class named Game1

The class named Game1 begins in [Listing 1](#).

Note:

Listing 1 . Beginning of the class named Game1.

```
namespace XNA0122Proj {
    public class Game1 :
Microsoft.Xna.Framework.Game {

        //Declare and populate instance variables
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        Texture2D myTexture;
        Vector2 spritePosition = new
Vector2(0.0f,0.0f);
        int slide = 8;//Used to move sprite across
screen.
        int scale = 4;//Size scale factor.
        int fast = 175;//Used for fast frame rate.
        int slow = 525;//Used for slow frame rate.
        int msPerFrame = 0;//Gets set for fast or
slow.
```

```

    int msElapsed;//Time since last new frame.
    int spriteCol;//Sprite column counter.
    int spriteColLim = 5;//Number of sprite
columns.
    int spriteRow;//Sprite row counter.
    int spriteRowLim = 2;//Number of sprite rows.
    int frameWidth;//Width of an individual image
    int frameHeight;//Height of an individual
image
    int xStart;//Corner of frame rectangle
    int yStart;//Corner of frame rectangle
    SpriteEffects noEffect = SpriteEffects.None;
    SpriteEffects flipEffect =
SpriteEffects.FlipHorizontally;
    SpriteEffects spriteEffect;//noEffect or
flipEffect
    int winWidth;//Width of the game window.
    int funSequenceCnt = 0;
    int pauseSequenceCnt = 0;

```

[Listing 1](#) declares a large number of instance variables that are used by code throughout the program. I will explain the purpose of the instance variables when we encounter them in the program code later.

The modified constructor for the Game1 class

The constructor for the **Game1** class is shown in [Listing 2](#).

Note:

Listing 2 . The constructor for the Game1 class.

```
public Game1() { //constructor
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";

    //Set the size of the game window.
    graphics.PreferredBackBufferWidth = 450;
    graphics.PreferredBackBufferHeight = 100;

} // end constructor
```

I added the last two statements to the standard constructor that is generated by Visual C# when you create a new project.

Although it isn't very clear in the [documentation](#), the values for **PreferredBackBufferWidth** and **PreferredBackBufferHeight** set the size of the game window provided that they are consistent with the screen resolution. These two statements caused the game window to be small as shown in [Figure 2](#) instead of the default size.

The overridden LoadContent method

The overridden **LoadContent** method is shown in [Listing 3](#).

Note:

Listing 3 . The overridden LoadContent method.

```
protected override void LoadContent() {
    //Create a new SpriteBatch object, which can
    be
    // used to draw textures.
    spriteBatch = new
    SpriteBatch(GraphicsDevice);
```



```

        //Load the image
        myTexture = Content.Load<Texture2D>
("dogcropped");

        //Initialize instance variables
        spriteCol = 0;
        spriteRow = 0;

        frameWidth = myTexture.Width / spriteColLim;
        frameHeight = myTexture.Height /
spriteRowLim;

        msPerFrame = fast;

        spriteEffect = flipEffect;

        winWidth = Window.ClientBounds.Width;
    }//end LoadContent

```

Load the image

The code to load the image, which is a sprite sheet, is the same as code that I have explained in earlier modules.

Initialization of variables

Some of the instance variables in [Listing 1](#) were initialized when they were declared. Others couldn't be initialized when they were declared for a variety of reasons.

Some could have been initialized in the constructor and others couldn't because the required information wasn't yet available.

I elected to initialize variables in the **LoadContent** method. By the time the **LoadContent** method executes, all of the information necessary to initialize the variables is available.

spriteCol and spriteRow

The variables named **spriteCol** and **spriteRow** will be used as counters to keep track of and to specify the column and row for a particular sprite image as shown in [Figure 1](#). The columns are numbered from 0 through 4 (five columns) and the rows are numbered from 0 through 1 (two rows).

frameWidth and frameHeight

These two variables specify the width and the height of an individual sprite image (see [Figure 1](#)). The width and the height of the individual sprite images are computed by dividing the total width of the image loaded in [Listing 3](#) by the number of sprite images in a row and by dividing the total height of the image loaded in [Listing 3](#) by the number of sprite images in a column.

msPerFrame

[Listing 1](#) declares two variables named **fast** and **slow** and initializes their values to 175 milliseconds and 525 milliseconds respectively. These two values are used to switch the animation frame rate between a fast rate and a slow rate by assigning one or the other value to the variable named **msPerFrame**. The **fast** value is assigned to **msPerFrame** in [Listing 3](#) to specify a fast frame rate when the animation begins.

spriteEffect

The **SpriteEffects** enumeration lists the following effects that can be applied to a sprite when it is drawn:

- FlipHorizontally
- FlipVertically
- None

The images in the raw sprite sheet shown in [Figure 1](#) are all facing to the left. The **FlipHorizontally** enumeration value will be used to cause the images to face to the right when the dog is moving from left to right across the game window. The **None** enumeration value will be used to cause the images to face to the left (the default) when the dog is moving from right to left across the game window.

This is accomplished with the variables named **spriteEffect** , **flipEffect** , and **noEffect** . The value of **spriteEffect** is initialized to **flipEffect** in [Listing 3](#) because the dog starts off moving from left to right.

winWidth

The variable named **winWidth** is set to the width of the game window. The value **Window.ClientBounds.Width** could have been used everywhere that **winWidth** is used but the length of the expression created some formatting problems when attempting to format the source code for this narrow publication format.

The overridden Update method

You will recall that after initialization, the XNA game loop switches back and forth between calling the **Update** method and the **Draw** method. The **Update** method is overridden to implement the game logic and the **Draw** method is overridden to render the current state of the game on the computer screen.

Note: There are some subtle timing issues that I explained in earlier modules and I won't get into them here.

The update method is fairly complex

The **Update** method in the earlier modules has been fairly simple. That is not the case in this module. The **Update** method in this module, which begins in [Listing 4](#), contains some fairly complex logic.

Note:

Listing 4 . Beginning of the Update method.

```
protected override void Update(GameTime
gameTime) {
    // Allows the game to exit

if(GamePad.GetState(PlayerIndex.One).Buttons.Back
==
ButtonState.Pressed)
    this.Exit();
    //-----
-----//

    //New code begins here.
    msElapsed +=
gameTime.ElapsedGameTime.Milliseconds;
    if(msElapsed > msPerFrame){
        //Reset the elapsed time and draw the new
frame.
        msElapsed = 0;
```

The animation frame rate

The code at the beginning of [Listing 4](#) is the standard code that is generated by Visual C# when you create a new Windows Game project.

The new code in [Listing 4](#) deals with the *animation frame rate* . The animation frame rate needs to be much slower than the default repetition rate of the game loop, which is 60 iterations per second. Otherwise the dog would run around so fast that it wouldn't look natural.

Many drawings will be repeated

Therefore, we won't change the drawing parameters during every iteration of the game loop. Instead, we will cause the sprite to be drawn in the game window sixty times per second, but many of those drawings will look exactly like the previous drawing.

We will accomplish this by changing the drawing parameters only once every **msPerFrame** milliseconds. (Recall that **msPerFrame** can have either of two values: **fast** and **slow** .)

The **GameTime** parameter

Each time the **Update** method is called, an incoming parameter contains information in an object of type **GameTime** that allows us to determine the number of milliseconds that have elapsed since the last time the **Update** method was called.

The documentation for the [GameTime](#) class has this to say:

Note: "Snapshot of the game timing state expressed in values that can be used by variable-step (real time) or fixed-step (game time) games."

The **ElapsedGameTime** property

The **GameTime** object has several properties, one of which is named [ElapsedGameTime](#) . This property, which is a structure of type **TimeSpan**

provides:

Note: "The amount of elapsed game time since the last update."

The `TimeSpan` structure

A [TimeSpan](#) structure has a large number of properties including one named [Milliseconds](#). This property:

Note: "Gets the milliseconds component of the time interval represented by the current `TimeSpan` structure."

Therefore, the first line of new code in [Listing 4](#) returns the elapsed time in milliseconds since the last call to the **Update** method.

Accumulate and compare elapsed time

[Listing 4](#) adds the value in milliseconds to an accumulator variable named **msElapsed** each time the **Update** method is called.

[Listing 4](#) also compares the accumulated value with the desired animation interval stored in **msElapsed**. If the accumulated value exceeds the desired animation interval, the accumulated value is set to zero and the body of the **if** statement that begins in [Listing 4](#) is executed to modify the drawing parameters.

Compute the location of the sprite to draw

The code in [Listing 5](#) computes the location in pixel coordinates of the sprite image that needs to be drawn the next time the **Draw** method is called.

Note:

Listing 5 . Compute the location of the sprite to draw.

```
xStart = spriteCol * frameWidth;  
yStart = spriteRow * frameHeight;
```

That sprite image is identified by the intersection of the **spriteCol** column and the **spriteRow** row in [Figure 1](#).

The column and row values are used in conjunction with the width and height of the sprite images to compute the coordinates of the upper-left corner of the sprite image to be drawn. These values are stored in **xStart** and **yStart** , which will be used in the **Draw** method to select the correct image from [Image 1](#) and to draw that image.

The overall animation cycle

The program plays five animation cycles of the five sprite images in the top row of [Figure 1](#). These five cycles are played with the **fast** animation frame rate discussed earlier. This is controlled by a counter variable named **funSequenceCnt** . (This name was chosen because these images portray the dog running and jumping and having fun.)

Two animation cycles from the bottom row of sprite images

Then the program plays two animation cycles of the five sprite images in the bottom row of [Figure 1](#). These five cycles are played with the **slow**

animation frame rate discussed earlier.

Pause and animate in the same location

During this period, the dog doesn't move across the game window but rather the animation cycles are played with the dog remaining in the same location. This is controlled by a counter variable named **pauseSequenceCnt** . (This name was chosen because the dog pauses and animates in the same location.)

After that, the overall cycle repeats.

Some complex logic

This is where the logic becomes a little complex and it remains to be seen how well I can explain it. However, my students are supposed to have the prerequisite knowledge that prepares them to dissect and understand complex logic directly from source code.

Adjust column and row counters

The drawing parameters have already been established to identity the sprite image that will be drawn the next time the **Draw** method is called. The code that follows is preparing for the sprite selection that will take place after that one.

Increment the column counter and compare

[Listing 6](#) increments the column counter and compares it with the number of columns in the sprite sheet in [Figure 1](#) . If they match, [Listing 6](#) resets the column counter to 0 and increments the **funSequenceCnt** to indicate that another one of the five cycles through the five images in the top row of [Figure 1](#) has been completed.

Note:

Listing 6 . Adjust column and row counters.

```
if(++spriteCol == spriteColLim){
    //Column limit has been hit, reset the
    // column counter and increment the
    // funSequenceCnt.
    spriteCol = 0;

    funSequenceCnt++;
```

Execute the pause sequence if it is time for it

The last statement in [Listing 6](#) increments the **funSequenceCnt** . The first statement in [Listing 7](#) tests to see if it has a value of 5. If so, all five cycles of the fun sequence have been executed and the code in the body of the **if** statement that begins at the top of [Listing 7](#) will be executed. The purpose of this code is to execute two cycles of the pause sequence.

Note:

Listing 7 . Execute the pause sequence if it is time for it.

```
if((funSequenceCnt == 5) || (spriteRow
== 1)){
    spriteRow = 1;//advance to second row
    //Increment the pause sequence
counter.
    pauseSequenceCnt++;
    //After two cycles in the pause mode,
reset
    // variables and start the overall
cycle
    // again.
```

```
        if(pauseSequenceCnt == 3){
            spriteRow = 0;
            funSequenceCnt = 0;
            pauseSequenceCnt = 0;
        } //end if on pauseSequenceCnt
    } //end if on funSequenceCnt

} //end if on spriteCollim in
```

The conditional clause

The conditional clause in the **if** statement at the top of [Listing 7](#) also tests to see if the row counter is pointing to row 1. If so, this means that the pause cycle has already begun and should be continued. Therefore, the body of that **if** statement will be executed. In other words, the body of the **if** statement will be executed if the **funSequenceCnt** is equal to 5 or the **spriteRow** is equal to 1.

Set the row counter to 1

The first statement in the body of the **if** statement sets the row counter to 1. This is necessary because control may have just entered the body of the **if** statement for the first time following completion of five cycles using the sprites in row 0 (the top row in [Figure 1](#)).

Increment the pauseSequenceCnt

Then [Listing 7](#) increments the **pauseSequenceCnt** and compares it with the literal value 3. If there is a match, two cycles of animation using the sprite images in the bottom row of [Figure 1](#) have been completed and it's time to return to the five cycles using the sprite images in the top row of [Figure 1](#).

To accomplish this, the row counter, the **funSequenceCnt** , and the **pauseSequenceCnt** are all set to 0. This will cause five cycles using the sprite images in the top row of [Figure 1](#) to be executed before control will once again enter the code in [Listing 7](#).

Adjust sprite position and frame rate

The code that we have examined so far mainly deals with selecting the sprite image to draw each time the **Draw** method is called. We haven't dealt with the location where the sprite will be drawn in the game window, the orientation of the sprite when it is drawn, and the frame animation rate of **fast** versus **slow** .

[Listing 8](#) adjusts these drawing parameters.

Note:

Listing 8 . Adjust sprite position and frame rate.

```
        if((spriteRow == 0) ||
           ((spriteRow == 1) && (spriteCol ==
0))) {
            msPerFrame = fast;
            spritePosition.X += frameWidth * scale /
slide;
        }
        else if ((spriteRow == 1) ||
                 ((spriteRow == 0) && (spriteCol
== 0))) {
            //Stop and display images.
            msPerFrame = slow;
        } //end if-else
```

More complex logic

The logic in [Listing 8](#) is fairly complex due mainly to the need to adjust the frame rate from fast to slow or from slow to fast when transitioning between the two rows of sprites in [Listing 1](#). Rather than to try to explain this logic, I am going to leave it as an exercise for the student to analyze the code and to determine where the frame rate transitions occur.

Scaling

Although I haven't mentioned it before, the **SpriteBatch.Draw** method (not the **Game.Draw** method) that will be used to draw the sprites in the game window has a scaling parameter that can be used to scale the images before drawing them.

[Listing 1](#) declares a variable named **scale** and sets its value to 4. This will be used as a scale factor when the sprite images are drawn.

The slide variable

[Listing 1](#) also declares a variable named **slide** and sets its value to 8. This variable is used to control how far the sprite moves each time it is drawn.

That distance, along with the new sprite position, is computed in [Listing 8](#) as the product of the width of the sprite image and the scale factor divided by the value of **slide**. This is a distance that I determined experimentally to cause the animation to look like I wanted it to look.

The sign of the variable named slide

It is worth noting that the sign of the variable named **slide** determines whether the incremental distance is positive or negative.

It is also worth noting that the change in **spritePosition.X** only occurs when sprites from the top row in [Figure 1](#) are being drawn. When sprites from the

bottom row in [Figure 1](#) are being drawn, the sprite is animated in place.

Move the sprite image back and forth across the game window

The code in [Listing 9](#) causes the sprite image to move back and forth across the game window always facing in the correct direction.

Note:

Listing 9 . Move the sprite image back and forth across the game window.

```
        if(spritePosition.X >
                                winWidth - frameWidth *
scale) {
            slide *= -1;
            spriteEffect = noEffect;
        }//end if

        if(spritePosition.X < 0){
            slide *= -1;
            spriteEffect = flipEffect;
        }//end if

    }//end if

    //-----
-----//
    //New code ends here.
    base.Update(gameTime);
} //end Update
```

Test for a collision with an edge

[Listing 9](#) tests for a collision between the sprite and the right edge or the left edge of the game window. If the sprite collides with the right edge, the sign on the variable named **slide** is changed to cause future incremental distance movements to be negative (from right to left).

If the sprite collides with the left edge, the sign on the variable named **slide** is changed to cause future incremental distance movements to be positive (from left to right).

The `spriteEffect` variable

In addition, when the sprite collides with one edge or the other, the value of the **spriteEffect** variable is set such that the dog will be facing the correct direction as it moves toward the other edge of the game window.

That concludes the explanation of the overridden **Update** method.

The overridden Draw method

The overridden **Draw** method selects the correct sprite image by extracting a rectangular area from the sprite sheet and draws the rectangle containing the sprite image at a specified location in the game window.

Note in [Figure 1](#) that the sprite sheet has a white non-transparent background. The game window is also caused to have a white background so that the white background of the rectangle containing the sprite image can't be distinguished from the game window background.

The overridden **Draw** method begins in [Listing 10](#).

Note:

Listing 10 . Beginning of the overridden Draw method.

```
        protected override void Draw(GameTime
gameTime) {

GraphicsDevice.Clear(Color.White); //Background
```

The statement in [Listing 10](#) erases everything in the game window by painting over it with a solid white background.

Begin the drawing process

You learned in an earlier module that the drawing process consists of a minimum of three statements.

The first statement is a call to the **SpriteBatch.Begin** method. This is followed by one or more calls to the **SpriteBatch.Draw** method. This is followed by a call to the **SpriteBatch.End** method.

There are four overloaded versions of the **SpriteBatch.Begin** method. The parameters for the different versions establish drawing parameters that apply to all of the subsequent calls to the **SpriteBatch.Draw** method until there is a call to the **SpriteBatch.End** method.

This program uses the simplest version of the **SpriteBatch.Begin** method with no parameters as shown in [Listing 11](#). This version simply:

Note: "Prepares the graphics device for drawing sprites."

Note:

Listing 11 . Begin the drawing process.

```
spriteBatch.Begin();
```

Call the `SpriteBatch.Draw` method

This program makes a single call to the **SpriteBatch.Draw** method followed by a call to the **SpriteBatch.End** method each time the **Game.Draw** method is called.

There are several overloaded versions of the **SpriteBatch.Draw** method and the one shown in [Listing 12](#) is one of the most complex. It was necessary to use this version to cause the sprite to be scaled by the value of the variable named **scale** when it is drawn.

Note:

Listing 12 . Call the `SpriteBatch.Draw` method.

```
spriteBatch.Draw(myTexture, //sprite sheet
                 spritePosition, //position
to draw
                 //Specify rectangular area
of the
                 // sprite sheet.
                 new Rectangle(
corner
                 xStart, //Upper left
                 yStart, // of rectangle.
height
                 frameWidth, //Width and
rectangle
                 frameHeight), // of
sprite
                 Color.White, //Don't tint
                 0.0f, //Don't rotate sprite
```



```

offset re //Origin of sprite. Can
// position above.
new Vector2(0.0f,0.0f),
//X and Y scale size scale
factor.
new Vector2(scale, scale),
spriteEffect, //Face
correctly
0); //Layer number
spriteBatch.End();

```

The nine parameters required for this version of the method are identified in the [documentation](#) as shown below. *(Note that square brackets were substituted for angle brackets in the following list to avoid problems in creating the cnxml format required for the OpenStax website.)*

- Texture2D texture
- Vector2 position
- Nullable[Rectangle] sourceRectangle
- Color color
- float rotation
- Vector2 origin
- Vector2 scale
- SpriteEffects effects
- float layerDepth

The first two parameters

The first two parameters that identify the sprite sheet and the position at which to draw the image are the same as the version that I explained in an earlier module.

Three interesting parameters

Three of the parameters shown in the above list and in [Listing 12](#) are parameters that are new to this module and in which I have an interest.

The remaining four parameters

Four of the parameters shown in the above list and in [Listing 12](#) are new to this module, but I don't have any interest in them. The program simply passes "harmless" values to them.

The rectangle

The first thing that is new to this module in [Listing 12](#) is the passing of a new object of type **Rectangle** as the third parameter.

With this version of the **Draw** method, only the contents of the sprite sheet that fall within that rectangle are drawn. The size and position of the rectangle are specified with:

- The coordinates of the upper left corner of the rectangle
- The width and the height of the rectangle

The upper left corner of the rectangle

The upper left corner of the rectangle is specified in [Listing 12](#) by the contents of the variables named **xStart** and **yStart**. The values in these two variables were assigned in [Listing 5](#). They specify the coordinates of the upper left corner of a small rectangle that contains one of the sprite images shown in [Figure 1](#).

The width and the height of the rectangle

The width and the height are specified by the contents of the variables named **frameWidth** and **frameHeight** . The values in these two variables were assigned in [Listing 3](#) as the width and height of the small rectangle that contains one of the sprite images in [Figure 1](#).

The horizontal and vertical scale factors

The seventh parameter requires a **Vector2D** object containing the scale factors to be applied to the horizontal and vertical sizes of the sprite before it is drawn. (You learned about the **Vector2D** class in an earlier module.)

A new object

[Listing 12](#) instantiates a new object of the **Vector2D** class that encapsulates the value stored in the variable named **scale** for the horizontal and vertical scale factors.

A value of 4 was assigned to the variable named **scale** when it was initialized in [Listing 1](#).

Four times larger

Therefore, the sprite images that are drawn in the game window are actually four times larger than the sprite images in the sprite sheet shown in [Figure 1](#) . (**Note that the images** were also scaled in [Figure 1](#) for display purposes in order to make them more visible. You need to take that into account if you download the sprite sheet and use it to replicate this program.)

Causing the sprite to face in the correct direction

The eighth parameter to the **Draw** method in [Listing 12](#) requires an object of type **SpriteEffects** . The contents of the variable named **spriteEffect** are

passed as this parameter. The contents of this variable were set to one of the following two values by the code in [Listing 9](#):

- noEffect - SpriteEffects.None
- flipEffect - SpriteEffects.FlipHorizontally

The purpose of this parameter is to cause the sprite to face in the correct direction.

The sprite needs to face to the left when it is moving from right to left. This is the default state of the sprite sheet shown in [Figure 1](#) so no flip is required.

The sprite needs to face to the right when it is moving from left to right. This requires a horizontal flip on the sprite images shown in [Figure 1](#).

Call the SpriteBatch.End method

The last statement in [Listing 12](#) calls the **SpriteBatch.End** method to terminate the drawing process for the current iteration of the game loop.

The visual frame rate

By default, the **Update** and **Draw** methods are each called approximately 60 times per second or approximately once every 16.67 milliseconds. This can be changed by program code but it was not changed in this program.

The fast frame rate

When the program is drawing the sprite images in the top row of [Figure 1](#), a new sprite image is selected for drawing only once every 175 milliseconds (see the variable named **fast**). Therefore, the same sprite image is drawn during ten or eleven successive iterations of the game loop.

The slow frame rate

When the program is drawing the sprite images in the bottom row of [Figure 1](#), a new sprite image is selected for drawing only once every 525 milliseconds (see the variable named **slow**). Therefore, each of the sprites in the bottom row is drawn during about 33 successive iterations of the game loop.

Avoid flicker but animate more slowly

By drawing the sprite images 60 times per second, the image can be maintained on the computer screen with no visible flicker. By drawing the same image multiple times in succession, the overall visual frame rate can be slowed down to produce a pleasing animation effect.

The end of the program

[Listing 13](#) shows the required call to the superclass of the **Draw** method, the end of the **Draw** method, the end of the class, and the end of the namespace.

Note:

Listing 13 . The end of the program.

```
        //Required standard code.  
        base.Draw(gameTime);  
    } //end Draw method  
} //End class  
} //End namespace
```

Run the program

I encourage you to copy the code from [Listing 14](#). Use that code to create an XNA project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do. Be sure to take the scale factor into account as mentioned [earlier](#).

Run my program

Click [here](#) to download a zip file containing my version of the program. Extract the folder named **XNA0122Proj** from the zip file and save it somewhere on your disk. Start **Visual C# 2010 Express** and select **Open Project...** from the **File** menu. Navigate to the project folder and select the file with the extension of **.sln**. This should cause the project to open and be ready to run or debug as described in the earlier module titled [Getting Started](#).

Summary

Among other things, you learned:

- How to create frame animation using a sprite sheet.
- How to flip and scale sprite images when they are drawn.
- How to implement different animation frame rates in the same program.
- How to work with different groups of sprite images in the same program.
- How to change the size of the game window.
- How to create a Rectangle object.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Xna0122-Frame Animation using a Sprite Sheet
- File: Xna0122.htm
- Published: 02/28/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the XNA program discussed in this module is provided in [Listing 14](#).

Note:

Listing 14 . Class Game1 from the project named XNA0122Proj.

```

/*Project XNA0122Proj
R.G.Baldwin, 12/28/09
Animation demonstration. Animates a dog running,
jumping,
and stopping to ponder and scratch the ground.
Uses two
different frame rates and a 5x2 sprite sheet. Runs
back
and forth across the game window always facing in
the
right direction.

```

```

*****
*****/

```

```

using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

```

```

namespace XNA0122Proj {
    public class Game1 :
Microsoft.Xna.Framework.Game {

```

```

        //Declare and populate instance variables
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        Texture2D myTexture;
        Vector2 spritePosition = new
Vector2(0.0f,0.0f);
        int slide = 0; //Used to move sprite across

```



```

    int slide = 0; //Used to move sprite across
screen.
    int scale = 4; //Size scale factor.
    int fast = 175; //Used for fast frame rate.
    int slow = 525; //Used for slow frame rate.
    int msPerFrame = 0; //Gets set for fast or
slow.
    int msElapsed; //Time since last new frame.
    int spriteCol; //Sprite column counter.
    int spriteColLim = 5; //Number of sprite
columns.
    int spriteRow; //Sprite row counter.
    int spriteRowLim = 2; //Number of sprite rows.
    int frameWidth; //Width of an individual image
    int frameHeight; //Height of an individual
image
    int xStart; //Corner of frame rectangle
    int yStart; //Corner of frame rectangle
    SpriteEffects noEffect = SpriteEffects.None;
    SpriteEffects flipEffect =

SpriteEffects.FlipHorizontally;
    SpriteEffects spriteEffect; //noEffect or
flipEffect
    int winWidth; //Width of the game window.
    int funSequenceCnt = 0;
    int pauseSequenceCnt = 0;

    public Game1() { //constructor
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        //Set the size of the game window.
        graphics.PreferredBackBufferWidth = 450;
        graphics.PreferredBackBufferHeight = 100;

    } // end constructor

```

```

protected override void Initialize() {
    //No special initialization needed
    base.Initialize();
} //end Initialize

protected override void LoadContent() {
    //Create a new SpriteBatch object, which can
be
    // used to draw textures.
    spriteBatch = new
SpriteBatch(GraphicsDevice);

    //Load the image
    myTexture = Content.Load<Texture2D>
("dogcropped");

    //Initialize instance variables
    spriteCol = 0;
    spriteRow = 0;

    frameWidth = myTexture.Width / spriteColLim;
    frameHeight = myTexture.Height /
spriteRowLim;

    msPerFrame = fast;

    spriteEffect = flipEffect;

    winWidth = Window.ClientBounds.Width;
} //end LoadContent

protected override void UnloadContent() {
    //No unload code needed.
} //end UnloadContent

protected override void Update(GameTime
gameTime) {

```

```

gameTime) {
    // Allows the game to exit

    if(GamePad.GetState(PlayerIndex.One).Buttons.Back
        ==
        ButtonState.Pressed)
        this.Exit();
        //-----
    -----//

    //New code begins here.

    //Compute the elapsed time since the last
new
    // frame. Draw a new frame only if this time
    // exceeds the desired frame interval given
by
    // msPerFrame
    msElapsed +=
gameTime.ElapsedGameTime.Milliseconds;
    if(msElapsed > msPerFrame){
        //Reset the elapsed time and draw the new
frame.
        msElapsed = 0;

        //Compute the location of the next sprite
to
        // draw from the sprite sheet.
        xStart = spriteCol * frameWidth;
        yStart = spriteRow * frameHeight;

        //Adjust sprite column and row counters to
        // prepare for the next iteration.
        if(++spriteCol == spriteColLim){
            //Column limit has been hit, reset the
            // column counter
            spriteCol = 0;
            //Increment the funSequenceCnt. The

```

```

//Increment the funSequenceCnt. The
program // plays five cycles of the fun sequence
with // the dog running and jumping using
sprites // from row 0 of the sprite sheet. Then
it // plays two cycles of the pause
sequence // using sprites from row 1 of the
sprite // sheet. Then the entire cycle repeats.
funSequenceCnt++;
if((funSequenceCnt == 5) || (spriteRow
== 1)){
    spriteRow = 1;//advance to second row
    //Increment the pause sequence
counter.
    pauseSequenceCnt++;
    //After two cycles in the pause mode,
reset
    // variables and start the overall
cycle
    // again.
    if(pauseSequenceCnt == 3){
        spriteRow = 0;
        funSequenceCnt = 0;
        pauseSequenceCnt = 0;
    }//end if
    }//end if
} //end if-else

//Adjust position of sprite in the output
window.

//Also adjust the animation frame rate
between
    // fast and slow depending on which set of

```

```

// fast and slow depending on which set of
five
// sprite images will be drawn.
if((spriteRow == 0) ||
    ((spriteRow == 1) && (spriteCol ==
0))) {
    msPerFrame = fast;
    spritePosition.X += frameWidth * scale /
slide;
}
else if ((spriteRow == 1) ||
    ((spriteRow == 0) && (spriteCol
== 0))) {
    //Stop and display images.
    msPerFrame = slow;
} //end if-else

//Cause the image to move back and forth
across
// the game window always facing in the
right
// direction.
if(spritePosition.X >
    winWidth - frameWidth *
scale) {
    slide *= -1;
    spriteEffect = noEffect;
} //end if

if(spritePosition.X < 0){
    slide *= -1;
    spriteEffect = flipEffect;
} //end if

} //end if

//-----
//

```

```

-----//
    //New code ends here.
    base.Update(gameTime);
} //end Update

protected override void Draw(GameTime
gameTime) {
GraphicsDevice.Clear(Color.White); //Background

    //Select the sprite image from a rectangular
area
    // on the sprite sheet and draw it in the
game
    // window. Note that this sprite sheet has a
white
    // non-transparent background.
    spriteBatch.Begin();

    spriteBatch.Draw(myTexture, //sprite sheet
        spritePosition, //position
to draw
        //Specify rectangular area
of the
        // sprite sheet.
        new Rectangle(
            xStart, //Upper left
corner
            yStart, // of rectangle.
            frameWidth, //Width and
height
            frameHeight), // of
rectangle
            Color.White, //Don't tint
sprite
            0.0f, //Don't rotate sprite
            //Origin of sprite. Can
offset re

```

```
offset =  
        // position above.  
        new Vector2(0.0f,0.0f),  
        //X and Y scale size scale  
factor.  
        new Vector2(scale,scale),  
        spriteEffect,//Face  
correctly  
        0);//Layer number  
        spriteBatch.End();  
  
        //Required standard code.  
        base.Draw(gameTime);  
    }//end Draw method  
}//End class  
}//End namespace
```

-end-

Xna0124-Using Background Images and Color Key Transparency
Learn how to display a sprite in front of a background image and how to cause the background image to change at runtime. Learn the difference between the position and origin parameters of the SpriteBatch.Draw method. Also learn how to deal with and use color key transparency.

Revised: Sun May 08 15:53:42 CDT 2016

This page is part of a Book titled [XNA Game Studio](#).

Table of Contents

- [Table of Contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [Layer depth](#)
 - [Color key transparency](#)
 - [Transparent green](#)
 - [Store your image in a lossless image file](#)
 - [Be aware of the default values](#)
- [Preview](#)
 - [The UFO image](#)
 - [Image of the planet](#)
 - [The screen output](#)
 - [Flying a relatively straight path](#)
 - [The background is moving](#)
 - [Preparing to change course](#)
 - [Flying towards the camera](#)

- [Heading for home](#)
- [Discussion and sample code](#)
 - [The class named Game1](#)
 - [The constructor for the class named Game1](#)
 - [Setting the size of the game window](#)
 - [Maintaining the aspect ratio](#)
 - [The overridden LoadContent method](#)
 - [The new material](#)
 - [Compute the position of the UFO relative to the game window](#)
 - [Compute the base scale factor for the background image](#)
 - [Not the only scale factor](#)
 - [An optical illusion](#)
 - [The overridden Update method](#)
 - [Controlling the animation speed](#)
 - [Draw new material every 83 milliseconds](#)
 - [Apply dynamic scaling to the background image](#)
 - [Two components in the scale factor](#)
 - [An optical illusion](#)
 - [The dynamic portion of the overall scale factor](#)
 - [Changing the origin of the background image](#)
 - [Increasing the scale alone is insufficient](#)
 - [Need to stabilize the location of the planet in the game window](#)
 - [Examples](#)
 - [The end of the Update method](#)

- [The overridden Game.Draw method](#)
 - [No need to clear the game window](#)
 - [Two main sections of code](#)
 - [Draw the background image showing the planet](#)
 - [Draw the UFO](#)
 - [Center the UFO image on the position](#)
 - [The UFO does not move](#)
 - [The front of the z-order stack](#)
 - [The end of the program](#)
 - [Recap on the origin and position parameters](#)
- [Run the program](#)
 - [Run my program](#)
 - [Summary](#)
 - [Miscellaneous](#)
 - [Complete program listing](#)

Preface

This module is one in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

An earlier module titled [Getting Started](#) provided information on how to get started programming with Microsoft's XNA Game Studio.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Property settings for green transparency.
- [Figure 2](#). The UFO image.
- [Figure 3](#). Reduced version of the alien planet image.
- [Figure 4](#). The UFO approaching the planet from some distance away.
- [Figure 5](#). Flying over the planet.
- [Figure 6](#). Flying over the planet.
- [Figure 7](#). Flying over the planet.
- [Figure 8](#). Preparing to change course.
- [Figure 9](#). Flying towards the camera very close to the surface of the planet.
- [Figure 10](#). Still flying towards the camera very close to the surface of the planet.
- [Figure 11](#). Almost touching the rings around the planet.
- [Figure 12](#). Heading for home.

Listings

- [Listing 1](#). Beginning of the class named Game1.
- [Listing 2](#). The constructor for the class named Game1.
- [Listing 3](#). Beginning of the overridden LoadContent method.
- [Listing 4](#). Compute the position of the UFO relative to the game window.
- [Listing 5](#). Compute the base scale factor for the background image.
- [Listing 6](#). Beginning of the overridden Update method.
- [Listing 7](#). Apply dynamic scaling to the background image.
- [Listing 8](#). Adjust the origin for the background image.
- [Listing 9](#). Beginning of the overridden Game.Draw method.
- [Listing 10](#). Draw the UFO.
- [Listing 11](#). The class named Game1 for the project named XNA0124Proj.

General background information

Layer depth

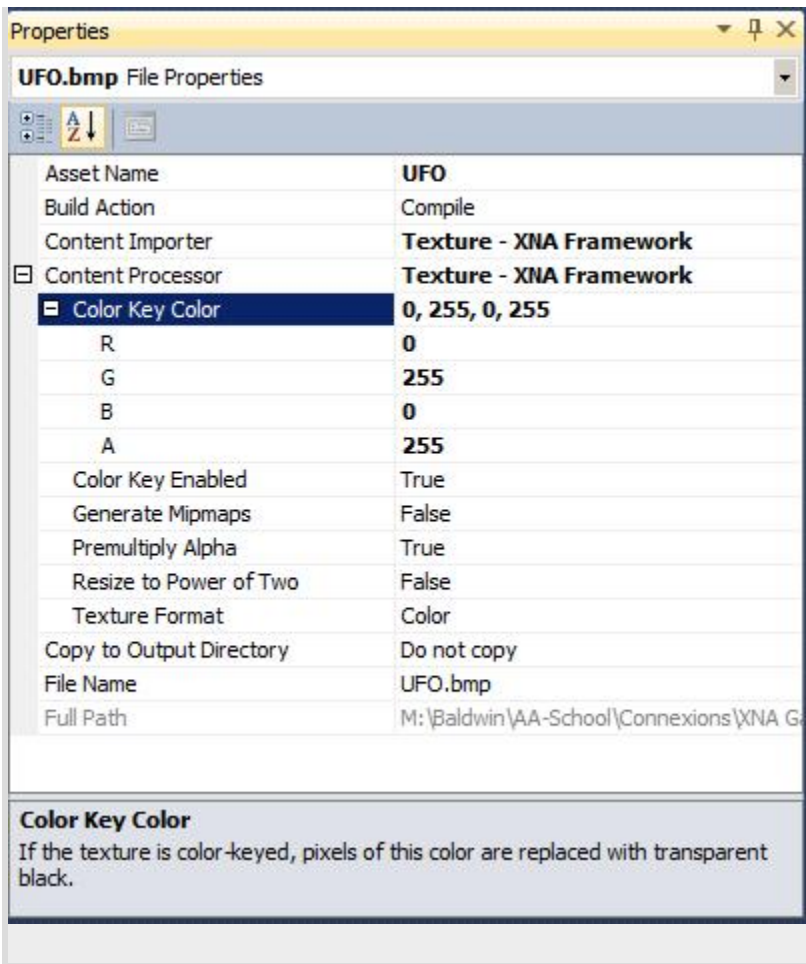
When you draw two or more sprites using the **SpriteBatch.Draw** method, you can specify the *z-order* as the last ([layerDepth](#)) parameter to the **Draw** method. By default, a non-transparent sprite that is drawn with a **layerDepth** value of 0.0 will hide sprites drawn with greater **layerDepth** values. The greater the **layerDepth** value, the further back will be the sprite in the *z-order*, up to a maximum value of 1.0, which represents the extreme back of the *z-order*.

Color key transparency

During the design phase of project development, you can cause the program to interpret one specific color as being transparent by setting the **Color Key Color** property to that color and setting the **Color Key Enabled** property to true as shown in [Figure 1](#) . Although I have never tried to do it, you can also apparently accomplish this at runtime by setting properties of the [TextureProcessor](#) class.

Note:

Figure 1 . Property settings for green transparency.



Transparent green

[Figure 1](#) shows the property settings required to cause every pixel having a color of pure green (0,255,0) to be replaced with transparent black, regardless of the actual alpha value of the green pixel.

Store your image in a lossless image file

If you use this capability, you must be careful not to store your image in a file that uses lossy compression, such as a JPEG file. If you do, the pure color that went into the file is not likely to be pure when you later extract

the image from the file. Instead, you should store your image in a lossless file such as a BMP file or a PNG file.

Be aware of the default values

Even if you don't plan to use this capability, you need to be aware of it. It seems that any time you add an existing image file to the content folder, the **Color Key Enabled** property will be true by default and the **Color Key Color** property value will be magenta (255,0,255). If you fail to disable the **Color Key Enabled** property, all of your pure magenta pixels will be replaced by transparent black pixels.

Note: In the early days of computer graphics, magenta was the defacto standard transparency color. It was referred to as "magic pink."

Preview

In this module, I will present and explain an animated sequence in which a UFO flies over an alien planet. You will view the action as if the camera is trained on the UFO while maintaining a constant position relative to the UFO.

As the program runs, the planet gets larger and larger creating the illusion that the UFO is getting closer and closer to the surface of the planet. Finally, the program resets and the sequence repeats.

The UFO image

The UFO image is shown in [Figure 2](#). Note the green background. All of the green pixels in the UFO image will be replaced by transparent black pixels when the program runs. I took advantage of the **Color Key**

transparency feature by setting the properties for the UFO image as shown in [Figure 1](#).

Note:

Figure 2 . The UFO image.

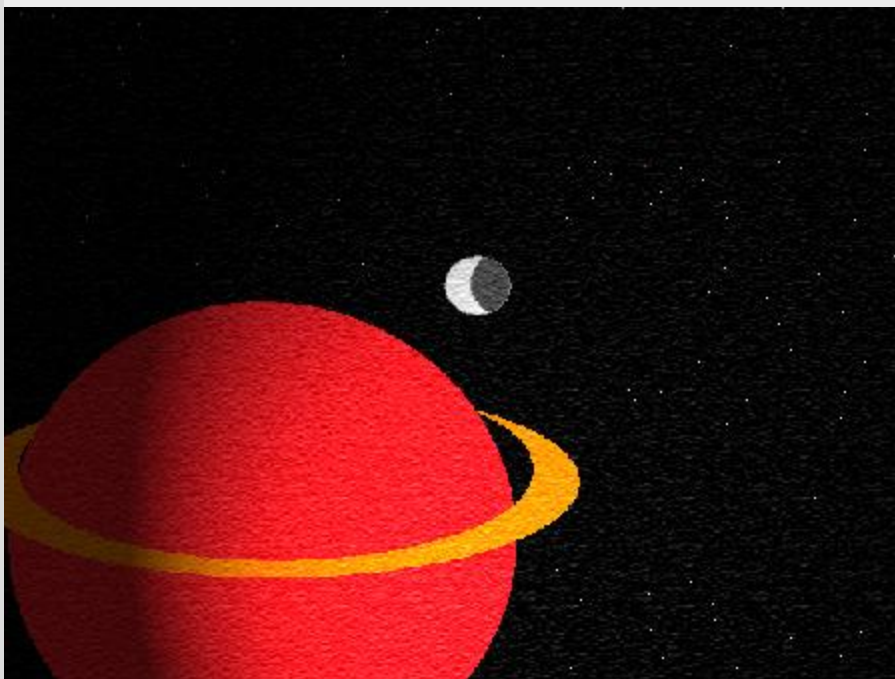


Image of the planet

[Figure 3](#) shows a reduced view of the image showing the planet that I used as a background image.

Note:

Figure 3 . Reduced version of the alien planet image.



The version that I used in the program was 640 pixels wide and 480 pixels high. However, that is too large to publish comfortably in this narrow publication format, so I reduced the image in [Figure 3](#) to 450x338 for publication purposes only. The source code that you will see later is based on an image size of 640x480.

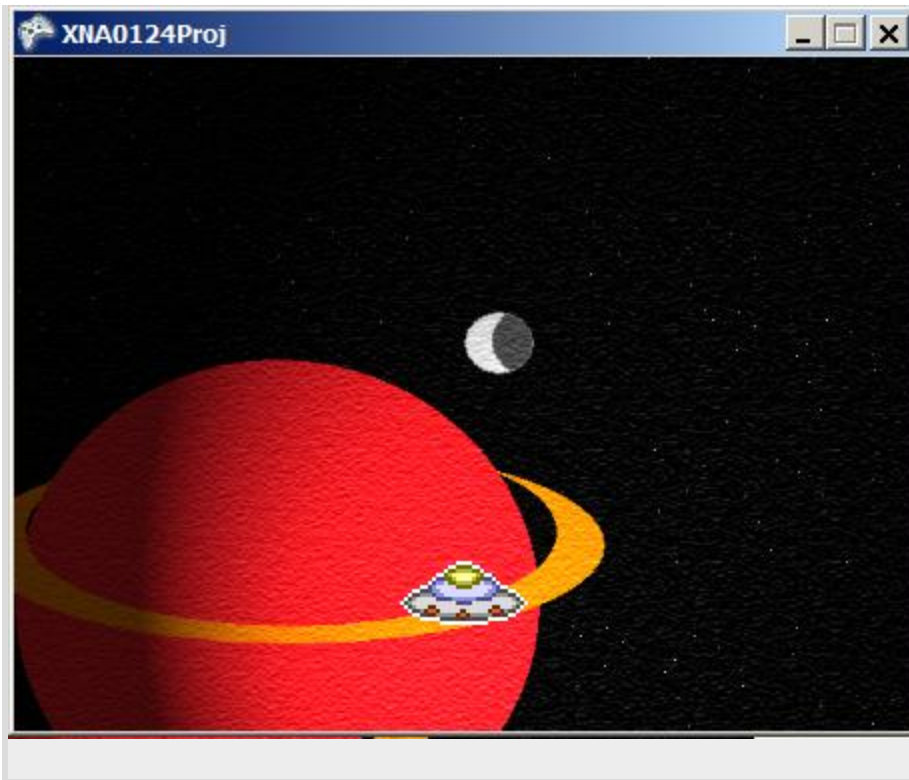
The screen output

[Figure 4](#) through [Figure 12](#) show nine screen shots taken while the animation was running.

In [Figure 4](#), the UFO is approaching the planet from some distance away. Note that the green portion of the UFO image has become transparent. Also note the size of the planet and its moon for comparison with subsequent screen shots.

Note:

Figure 4 . The UFO approaching the planet from some distance away.



Flying a relatively straight path

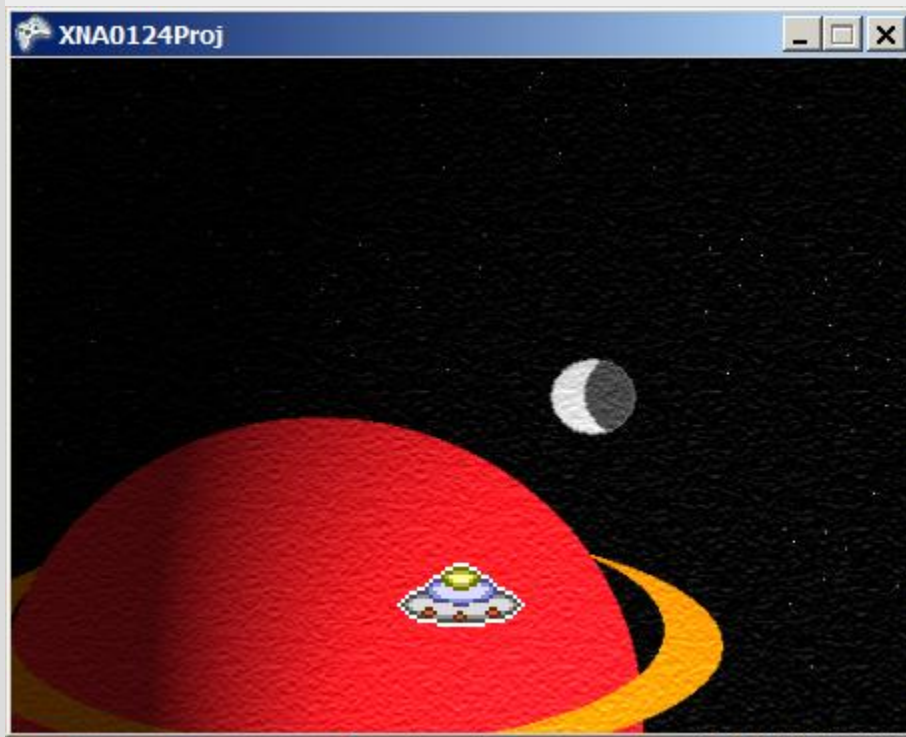
In [Figure 5](#) through [Figure 7](#), the UFO continues along a relatively straight path flying over the planet and getting closer to the surface of the planet all the time. Note how the planet increases in size from one image to the next, giving the illusion that the UFO is getting closer to the surface of the planet.

The background is moving

Note also that the position of the UFO in the game window is not changing. Instead an illusion of motion is created by causing the background to change. This is a technique that was used for many years in the movies to create the illusion that actors were in a car driving along the highway when in fact, the car was standing still inside a studio. The image of the stationary car was superimposed on a moving background image.

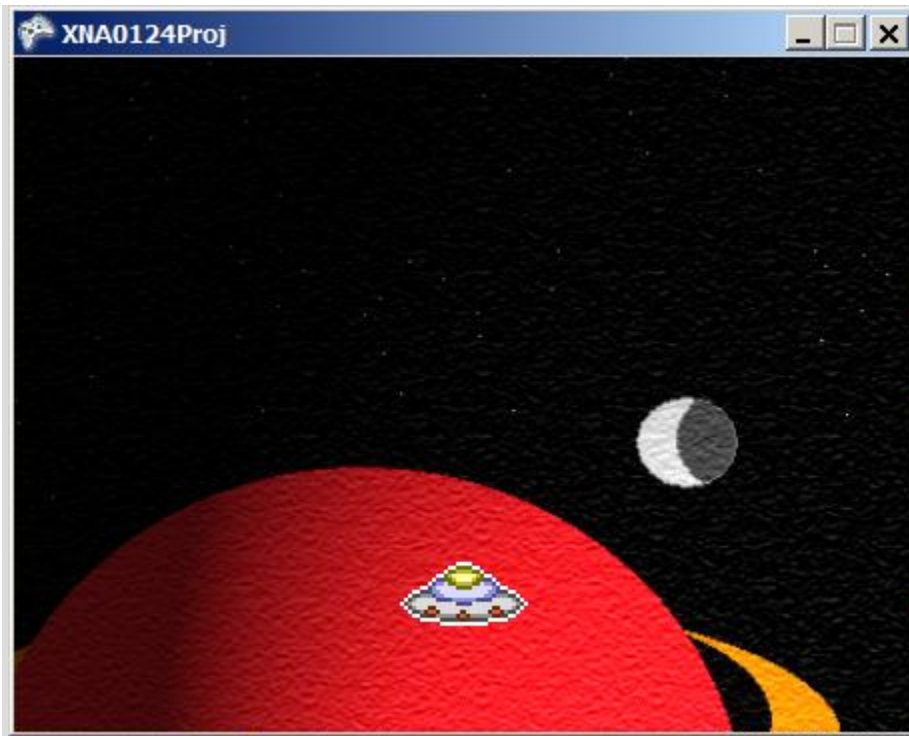
Note:

Figure 5 . Flying over the planet.

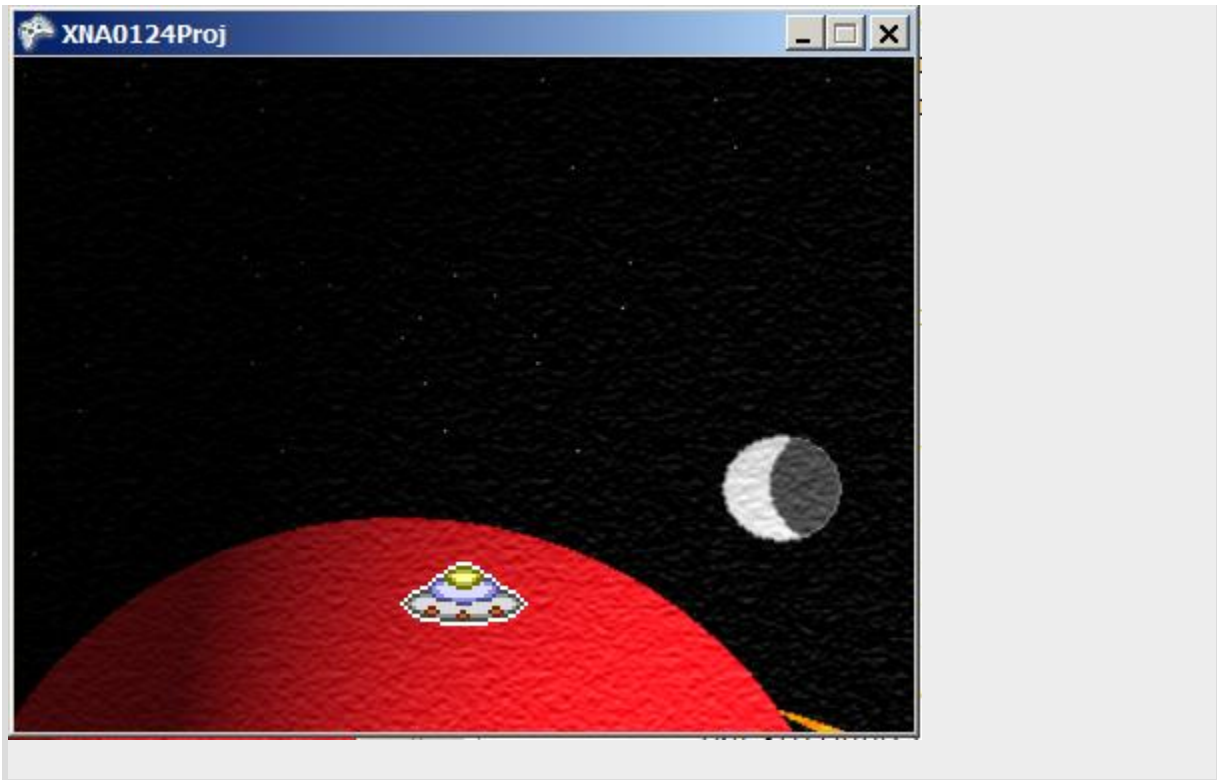


Note:

Figure 6 . Flying over the planet.



Note:
Figure 7 . Flying over the planet.

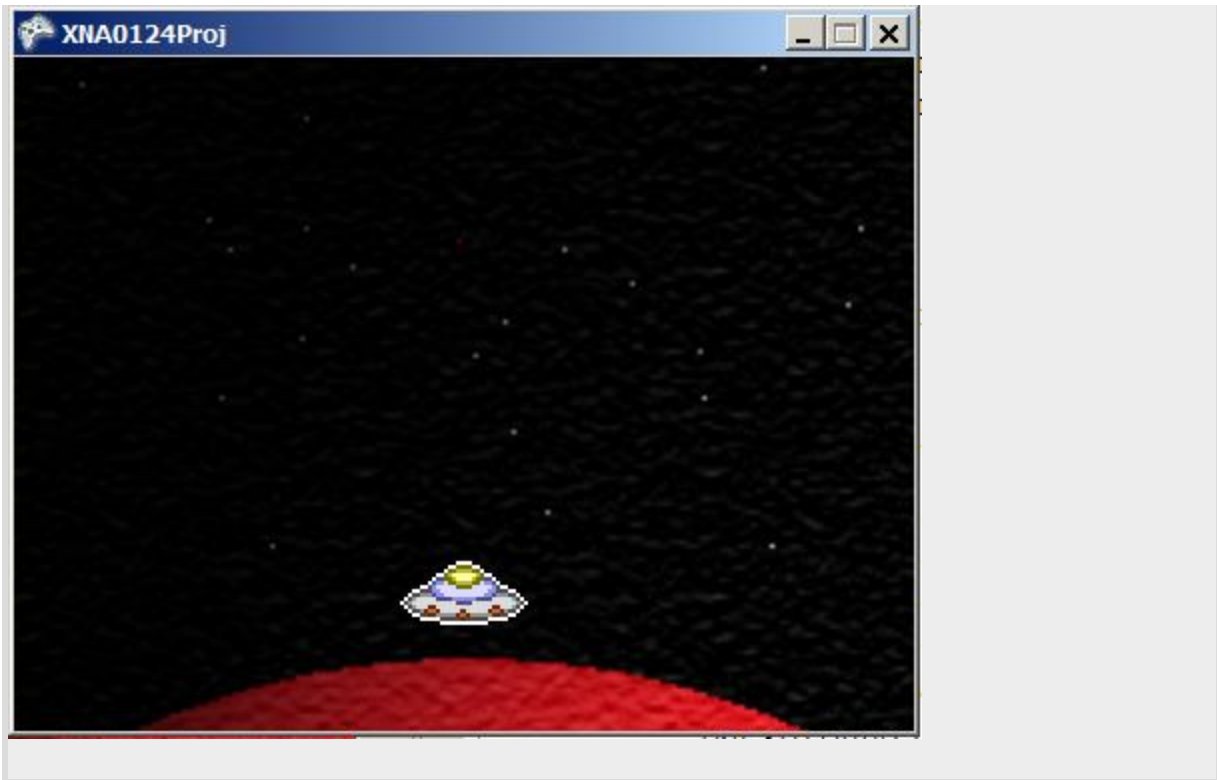


Preparing to change course

In [Figure 8](#), the UFO has gone about as far as it is going to go along the original course. It is preparing to change course and come back towards the camera getting ever closer to the surface of the planet.

Note:

Figure 8 . Preparing to change course.



Flying towards the camera

In [Figure 9](#) and [Figure 10](#), the UFO has changed course. It is now flying towards the camera very close to the surface of the planet. You can even see some surface features on the planet (due to the distortion that is produced when a bitmap image is enlarged).

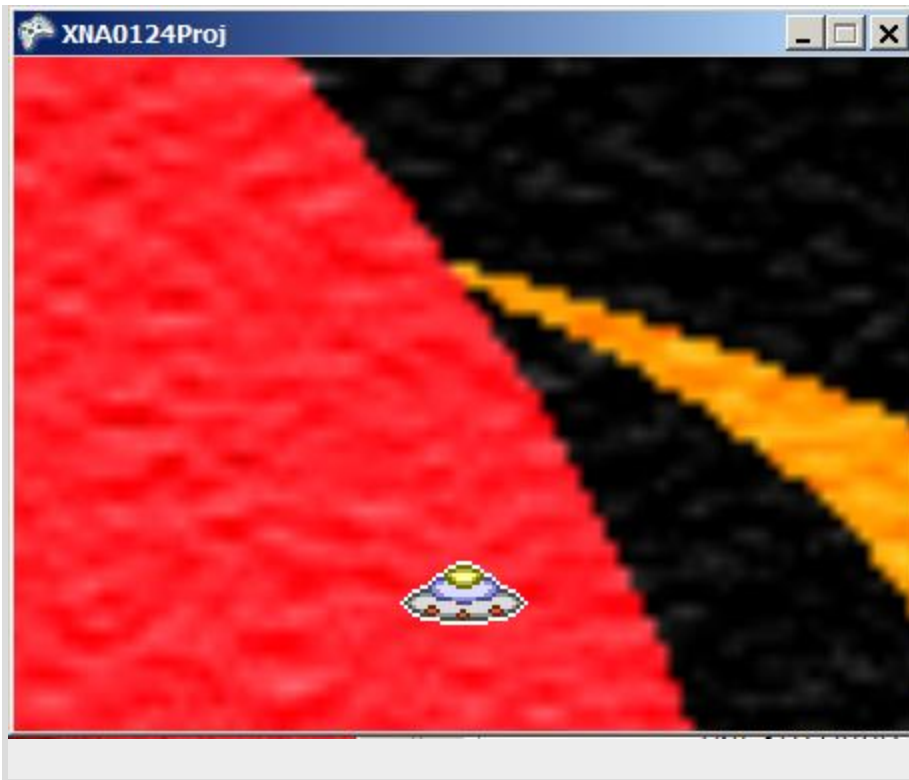
Note:

Figure 9 . Flying towards the camera very close to the surface of the planet.



Note:

Figure 10 . Still flying towards the camera very close to the surface of the planet.



In [Figure 11](#), the UFO is almost touching the rings around the planet.

Note:

Figure 11 . Almost touching the rings around the planet.

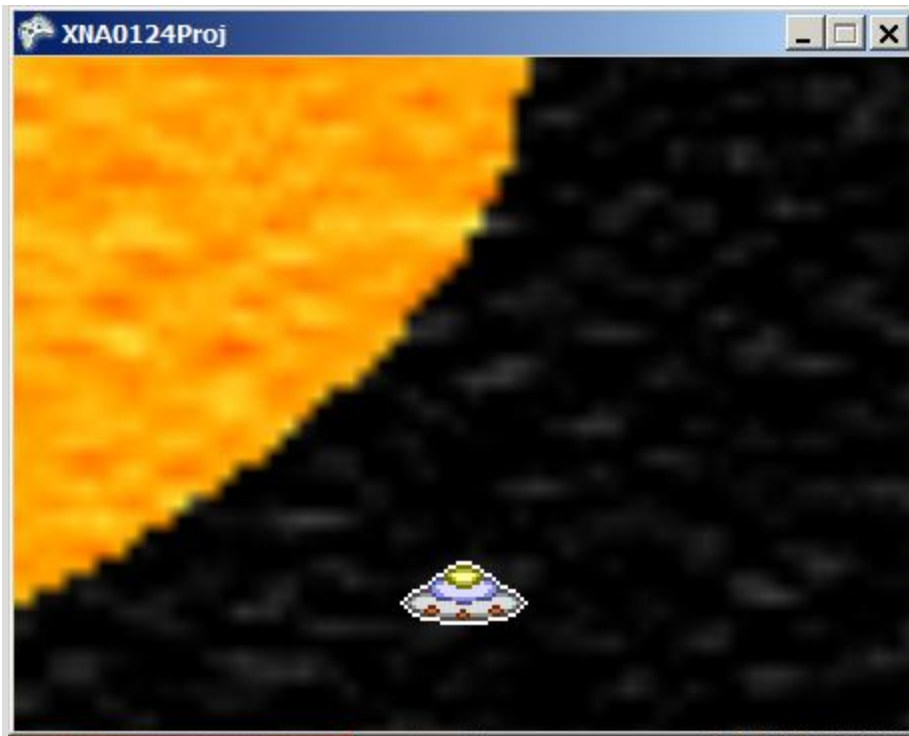


Heading for home

In [Figure 12](#), the UFO leaves the planet and heads for home. Shortly after this, the program will reset and repeat the animation sequence.

Note:

Figure 12 . Heading for home.



Discussion and sample code

As usual, I will explain the code in this program in fragments. A complete listing of the class named **Game1** is provided in [Listing 11](#) near the end of the module.

The class named Game1

The class named **Game1** begins in [Listing 1](#).

Note:

Listing 1 . Beginning of the class named Game1.

```

namespace XNA0124Proj {

    public class Game1 :
Microsoft.Xna.Framework.Game {

        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        private Viewport viewport;
        private Vector2 ufoPosition;
        private float backgroundScale;
        private float backgroundBaseScale;
        private float dynamicScale = 0.0f;
        int msElapsed;//Time since last new frame.
        int msPerFrame = 83;//30 updates per second
        Texture2D spaceTexture;//background image
        Texture2D ufoTexture;//ufo image
        Vector2 spaceOrigin;//origin for drawing
background

```

[Listing 1](#) contains the declarations of several instance variables that will be used later in the program. I will explain their purpose when they are used later.

The constructor for the class named Game1

The constructor for the class is shown in [Listing 2](#).

Note:

Listing 2 . The constructor for the class named Game1.

```

    public Game1() { //constructor
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        //Set the size of the game window, causing
the
        // aspect ratio of the game window to match
the
        // aspect ratio of the background image,
which is
        // 640 wide by 480 high.
        graphics.PreferredBackBufferWidth = 450;
        graphics.PreferredBackBufferHeight =
            (int)
(450.0*480/640);
    } //end constructor

```

Setting the size of the game window

The last two statements in [Listing 2](#) set the size of the game window. You have seen code like this before in an earlier module. The only thing new here is the expression used to compute the value for the height of the game window.

Maintaining the aspect ratio

The background image shown in [Figure 3](#) is 640 pixels wide by 480 pixels high. When it is initially drawn, it will be scaled to a width of 450 pixels keeping the same width to height ratio (the aspect ratio). The intent is that it will initially just fit inside the game window.

The expression used to compute the new height of the game window in [Listing 2](#) causes the aspect ratio of the game window to match the aspect ratio of the background image.

The overridden LoadContent method

The overridden **LoadContent** method begins in [Listing 3](#).

Note:

Listing 3 . Beginning of the overridden LoadContent method.

```
protected override void LoadContent() {  
    // Create a new SpriteBatch, which can be  
used to  
    // draw textures.  
    spriteBatch = new  
SpriteBatch(GraphicsDevice);  
  
    //Load the two images.  
    spaceTexture = Content.Load<Texture2D>  
("space");  
    ufoTexture = Content.Load<Texture2D>("ufo");  
  
    //Get a reference to the viewport.  
    viewport = graphics.GraphicsDevice.Viewport;
```

The new material

The only thing that is new in [Listing 3](#) is the last statement.

Viewport is a property of the [GraphicsDevice](#) class and is a [Structure](#) named **Viewport** .

The description in the [documentation](#) doesn't make a lot of sense to me. I believe it is safe to say, however, that in this XNA program, the viewport is synonymous with the game window. However, that is not always the case. A game window can contain more than one viewport. See [How To: Use Viewports for Split Screen Gaming](#) for an example.

Compute the position of the UFO relative to the game window

The code in [Listing 4](#) computes a position vector which, when applied during the drawing of the UFO, will cause the UFO to occupy a position approximately like that shown in [Figure 4](#).

Note:

Listing 4 . Compute the position of the UFO relative to the game window.

```
ufoPosition.X = viewport.Width / 2;  
ufoPosition.Y = viewport.Height - 70;
```

I say approximately because one additional adjustment to the position of the UFO will be made later using the **origin** parameter of the **SpriteBatch.Draw** method. If that adjustment were not made, the upper left corner of the rectangle that contains the UFO would be placed at the position computed in [Listing 4](#). The origin property will be used to center the UFO on the position that is computed in [Listing 4](#).

Compute the base scale factor for the background image

[Listing 5](#) computes a scale factor which, when applied to the background image during the drawing process, will cause the 640x480 background image to just fit the game window. Recall that the size of the game window was set in the constructor of [Listing 2](#) taking the aspect ratio of the background image into account.

Note:

Listing 5 . Compute the base scale factor for the background image.

```
        backgroundBaseScale = (float)(450.0 /  
640.0);  
    }//end LoadContent
```

Not the only scale factor

The scale factor computed in [Listing 5](#) will be applied to the background image when the program first starts running. However, during most of the running of the program, a different scale factor will be applied to the background image and the scale factor computed in [Listing 5](#) will be only one component of that overall scale factor.

An optical illusion

The application of a scale factor to the background image that changes with time is what causes the planet to grow in size giving the illusion that the UFO is approaching the surface of the planet.

[Listing 5](#) signals the end of the **LoadContent** method.

The overridden Update method

The overridden **Update** method begins in [Listing 6](#).

Note:

Listing 6 . Beginning of the overridden Update method.

```
        protected override void Update(GameTime  
gameTime) {  
  
            //Compute the elapsed time since the last
```

```

update.
    // Draw new data only if this time exceeds
the
    // desired frame interval given by
msPerFrame
    msElapsed +=
gameTime.ElapsedGameTime.Milliseconds;
    if(msElapsed > msPerFrame) {
        //Reset the elapsed time and draw the
frame with
        // new data.
        msElapsed = 0;

```

Controlling the animation speed

The code in [Listing 6](#) is not new to this module. I explained code like this in an earlier module.

Briefly, the **Update** method is called sixty times per second by default (I didn't change the default). That is too fast to produce an animation that looks the way I wanted it to look.

Draw new material every 83 milliseconds

The code in [Listing 6](#) causes the sprites to be drawn sixty times per second, but new material is drawn only once every **msPerFrame** milliseconds. The value for **msPerFrame** is set to 83 milliseconds in [Listing 1](#), but as you learned in an earlier module, its value could be changed by program code as the program is running.

Therefore, in this program, new material is drawn 30 times per second. Every other frame that is drawn looks exactly like the one before it, but the human eye is not fast enough to be able to detect that.

Apply dynamic scaling to the background image

A different and ever increasing scale factor is applied to the background image each time the body of the **if** statement in the **Update** method in [Listing 6](#) is executed.

Two components in the scale factor

This overall scale factor is composed of the fixed **backgroundBaseScale** factor computed in [Listing 5](#) and a dynamic scale factor named **dynamicScale** that is computed in [Listing 7](#).

An optical illusion

This is what causes the size of the planet to continue to increase during the animation sequence creating the illusion that the UFO is getting closer to the surface of the planet.

The dynamic portion of the overall scale factor

The value of the dynamic portion of the overall scale factor increases by 0.03 each time this code is executed. Once the dynamic portion reaches a value of 10, it is reset to zero and the animation sequence plays again from the beginning.

This is accomplished by the scaling algorithm shown in [Listing 7](#).

Note:

Listing 7 . Apply dynamic scaling to the background image.

```
factor    //Reset the animation if the dynamicScale
          // is greater than 10.
```



```

        if(dynamicScale > 10) {
            dynamicScale = 0.0f;
        } else {
            //Increase the dynamicScale factor and
use it
            // to compute a new scale factor that
will be
            // applied to the background image in
the next
            // call to the Draw method.
            dynamicScale += 0.03f;
            backgroundScale =
                backgroundBaseScale * (1 +
dynamicScale);
        } //end if-else on dynamicScale

```

Changing the origin of the background image

The **SpriteBatch.Draw** method has one parameter named **position** and another parameter named **origin** . You can cause the **origin** to be any point in the image relative to its upper left corner.

Note: Important: The **position** parameter specifies the position in the game window where the image's **origin** will be drawn.

If you hold the **position** parameter constant and vary the **origin** , the image will appear to slide around within the game window.

Increasing the scale alone is insufficient

Consider the background image shown in [Figure 3](#). Assume that the **origin** of the image is the upper left corner of the image. Also assume that the image will be drawn with that **origin** in the upper left corner of the game window.

If we were to scale the image to make it larger, the planet would be pushed down and to the right and would exit the game window near the bottom right corner of the game window. That is not what we are looking for.

Need to stabilize the location of the planet in the game window

To achieve the effect we are looking for, we need to cause the planet to remain in pretty much the same location within the game window as it gets larger. One way to accomplish that is by causing the origin to move down and to the right within the background image as the planet becomes larger. (Obviously that is not the only way to accomplish it.)

That effect is accomplished by the algorithm shown in [Listing 8](#).

Note:

Listing 8 . Adjust the origin for the background image.

```
        //Also use the dynamicScale factor to
compute a
        // new value for the origin of the
background.
        // The origin is the point in the image
that is
        // drawn at the point in the game window
that is
        // passed as the second (position)
parameter in
        // the call to the SpriteBatch.Draw
method. This
        // has the effect of causing the
```

```

background
    // image to slide up and to the left at
the same
    // time that it is getting larger.
    spaceOrigin =
        new Vector2((float)(450 * (dynamicScale)
/ 12),
                    (float)(338 * (dynamicScale)
/ 10));
    }//end if on msElapsed

    base.Update(gameTime);
} //end Update

```

Examples

For example, in [Figure 4](#), the origin of the image (the point drawn in the upper left corner of the game window) was out in space far removed from the planet. By [Figure 10](#) the origin had shifted to the surface of the planet causing a point on the surface of the planet to be drawn in the upper left corner of the game window. By [Figure 12](#), the origin had shifted all the way down and to the right to be on the surface of the ring that surrounds the planet.

The end of the Update method

[Listing 8](#) signals the end of the overridden **Update** method.

The overridden Game.Draw method

To avoid confusion, I want to remind you that when programming in XNA, you override the **Game.Draw** method. Within that method, you make calls

to the **SpriteBatch.Draw** method. These are entirely different methods belonging to different objects even though they have the same name.

No need to clear the game window

The overridden **Game.Draw** method begins in [Listing 9](#). Unlike the programs in earlier modules in this book, there is no need to clear the game window to a constant color in this program because the background image of the planet completely fills the game window.

Two main sections of code

In this program, the overridden **Game.Draw** method consists of two main sections of code. The first section consists of a **Begin , Draw , End** sequence with alpha blending turned off. This code is used to draw the background image. Alpha blending is turned off to prevent any areas of the background image that may have a low alpha value or any green pixels from appearing to be transparent.

The second section consists of another **Begin , Draw , End** sequence with alpha blending turned on. This code is used to draw the UFO. Alpha blending is turned on to cause the green areas in the UFO sprite shown in [Figure 2](#) to be transparent and to let the background image show through.

Draw the background image showing the planet

The first major section of code is shown in [Listing 9](#).

Note: Note the change that was made in the call to the **SpriteBatch.Begin** method to upgrade the program from XNA 3.1 to XNA 4.0.

Note:

Listing 9 . Beginning of the overridden Game.Draw method.

```
protected override void Draw(GameTime
gameTime) {

    // Turn off blending to draw the planet in
the
    // background. Note the update for XNA 4.0.
//    spriteBatch.Begin(SpriteBlendMode.None);

spriteBatch.Begin(SpriteSortMode.Immediate,
BlendState.Opaque);

    //Draw the background.
    spriteBatch.Draw(spaceTexture, //sprite
                                Vector2.Zero, //position re
window
                                null, //rectangle
                                Color.White, //tint
                                0, //rotation
                                spaceOrigin, //origin
                                backgroundScale, //scale
                                SpriteEffects.None,
                                1.0f); //layer, near the
back

    spriteBatch.End();
```

The only things that are new here are the interactions among the **position** parameter, the **origin** parameter, and the **scale** parameter that I explained earlier.

It is probably also worth noting that the last parameter has a value of 1.0, which causes this image to be displayed behind every other image.

Draw the UFO

The code in [Listing 10](#) draws the UFO image with alpha blending turned on to cause the green areas of the image to be transparent.

Note: Again, note the change that was made in the call to the **SpriteBatch.Begin** method to upgrade the program from XNA 3.1 to XNA 4.0.

Note:

Listing 10 . Draw the UFO.

```
// Turn on blending to draw the UFO in the
// foreground. Note the update for XNA 4.0.
//
spriteBatch.Begin(SpriteBlendMode.AlphaBlend);
    spriteBatch.Begin(SpriteSortMode.Deferred,
BlendState.AlphaBlend);
    //Center the UFO at the point in the window
    // identified as UFO position. The UFO image
is
    // 64x33 pixels.
    Vector2 origin = new Vector2(32,16);
    spriteBatch.Draw(ufoTexture, //sprite
                        ufoPosition, //position re
window
                        null, //rectangle
                        Color.White, //tint
                        0, //rotation
                        origin, //origin
                        1, //scale
                        SpriteEffects.None,
                        0f); //layer, 0 is in front
```

```
        spriteBatch.End();

        base.Draw(gameTime);
    }//end Draw method
} //end class
} //end namespace
```

Center the UFO image on the position

The UFO image is 64 pixels wide and 33 pixels high. The origin is set to the center of the image with coordinates of (32,16) relative to the upper left corner of the image.

Then the image is drawn with the **origin** at the **ufoPosition** , which was computed in [Listing 4](#).

The value of **ufoPosition** specifies a point that is in the horizontal center of the game window and 70 pixels from the bottom of the game window. Therefore, the center of the UFO is positioned at the horizontal center of the game window and 70 pixels up from the bottom of the game window as shown in [Figure 4](#) through [Figure12](#).

The UFO does not move

The UFO doesn't move. It stays in one place and the background image is animated behind it.

The front of the z-order stack

The value of the last parameter in [Listing 10](#) is zero. This places the UFO in front of every other image. Of course, in this case, there are only two images: the background image at the back and the UFO image at the front.

The end of the program

[Listing 10](#) signals the end of the overridden Draw method, the end of the class, and the end of the program.

Recap on the origin and position parameters

The **SpriteBatch.Draw** method has an **origin** parameter and a **position** parameter.

The **origin** parameter specifies a point somewhere in the image being drawn relative to the upper left corner of the image. The **position** parameter specifies where that point will be drawn relative to the upper left corner of the game window.

Run the program

I encourage you to download [Figure 2](#) and [Figure 3](#) and to copy the code from [Listing 11](#). Use that code to create an XNA project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do. Note that the image that you download from [Figure 3](#) will be smaller than the image that I used. Therefore, your scaling will probably need to be different.

Run my program

Click [here](#) to download a zip file containing my version of the program. Extract the folder named **XNA0124Proj** from the zip file and save it somewhere on your disk. Start **Visual C# 2010 Express** and select **Open Project...** from the **File** menu. Navigate to that folder and select the file with the extension of **.sln** . This should cause the project to open and be ready to run or debug as described in the earlier module titled [Getting Started](#).

Summary

You learned about the following major topics in this module:

- How to display a sprite in front of a background image.
- The difference between the **position** and **origin** parameters of the **SpriteBatch.Draw** method.
- How to cause the background image to change at runtime.
- How to deal with and use color key transparency.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Xna0124-Using Background Images and Color Key Transparency
- File: Xna0124.htm
- Published: 02/28/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales

nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I: I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the XNA program discussed in this module is provided in [Listing 11](#). Note the changes that were made in the calls to the **SpriteBatch.Begin** method to upgrade the program from XNA 3.1 to XNA 4.0. Blending was turned off in the first call just in case there were pixels in the background image having a color that matches the key color value.

Note:

Listing 11 . The class named Game1 for the project named XNA0124Proj.

```
/*Project XNA0124Proj
 * Illustrates displaying a sprite with color key
 * transparency in front of a background image.
 * Must modify the color key property value for
the ufo
 * sprite, changing the key color from the default
of
 * 255,0,255 (magenta or magic pink) to 0,255,0
(green).
 * The scale and the origin of the background
image is
 * changed over time giving the illusion of a ufo
 * flying over a planet.
*****
*****/
```

```

using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace XNA0124Proj {

    public class Game1 :
Microsoft.Xna.Framework.Game {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        private Viewport viewport;
        private Vector2 ufoPosition;
        private float backgroundScale;
        private float backgroundBaseScale;
        private float dynamicScale = 0.0f;
        int msElapsed;//Time since last new frame.
        int msPerFrame = 83;//30 updates per second
        Texture2D spaceTexture;//background image
        Texture2D ufoTexture;//ufo image
        Vector2 spaceOrigin;//origin for drawing
background

        public Game1() { //constructor
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";

            //Set the size of the game window, causing
the
            // aspect ratio of the game window to match
the
            // aspect ratio of the background image,
which is
            // 640 wide by 480 high.
            graphics.PreferredBackBufferWidth = 450;
            graphics.PreferredBackBufferHeight =
                (int)
(450.0*480/640);

```

```

    }//end constructor
    //-----
-----//

    protected override void Initialize() {
        // No initialization required
        base.Initialize();
    }//end Initialize
    //-----
-----//

    protected override void LoadContent() {
        // Create a new SpriteBatch, which can be
used to
        // draw textures.
        spriteBatch = new
SpriteBatch(GraphicsDevice);

        //Load the two images.
        spaceTexture = Content.Load<Texture2D>
("space");
        ufoTexture = Content.Load<Texture2D>("ufo");

        //Get a reference to the viewport.
        viewport = graphics.GraphicsDevice.Viewport;

        //Compute the position of the ufo relative
to the
        // game window.
        ufoPosition.X = viewport.Width / 2;
        ufoPosition.Y = viewport.Height - 70;

        //Set the backgroundBaseScale factor such
that
        // the entire background image will fit in
the
        // game window. Note that the aspect ratio

```

```

of the
    // game window was set to match the aspect
ratio
    // of the background image in the
constructor.
    backgroundBaseScale = (float)(450.0 /
640.0);
    }//end LoadContent
    //-----
-----//

    protected override void UnloadContent() {
        // No unload required
    }//end UnloadContent
    //-----
-----//

    protected override void Update(GameTime
gameTime) {

        //Compute the elapsed time since the last
update.
        // Draw new data only if this time exceeds
the
        // desired frame interval given by
msPerFrame
        msElapsed +=
gameTime.ElapsedGameTime.Milliseconds;
        if(msElapsed > msPerFrame) {
            //Reset the elapsed time and draw the
frame with
            // new data.
            msElapsed = 0;

            //Reset the animation if the dynamicScale
factor
            // is greater than 10.

```

```

        if(dynamicScale > 10) {
            dynamicScale = 0.0f;
        } else {
            //Increase the dynamicScale factor and
use it
            // to compute a new scale factor that
will be
            // applied to the background image in
the next
            // call to the Draw method.
            dynamicScale += 0.03f;
            backgroundScale =
                backgroundBaseScale * (1 +
dynamicScale);
        } //end if-else on dynamicScale

        //Also use the dynamicScale factor to
compute a
        // new value for the origin of the
background.
        // The origin is the point in the image
that is
        // drawn at the point in the game window
that is
        // passed as the second (position)
parameter in
        // the call to the SpriteBatch.Draw
method. This
        // has the effect of causing the
background
        // image to slide up and to the left at
the same
        // time that it is getting larger.
        spaceOrigin =
            new Vector2((float)(450 * (dynamicScale)
/ 12),
                        (float)(338 * (dynamicScale)

```

```

/ 10));
    }//end if on msElapsed

    base.Update(gameTime);
} //end Update
//-----
-----//

protected override void Draw(GameTime
gameTime) {

    // Turn off blending to draw the planet in
the
    // background. Note the update for XNA 4.0.
//    spriteBatch.Begin(SpriteBlendMode.None);

spriteBatch.Begin(SpriteSortMode.Immediate,
BlendState.Opaque);

    //Draw the background.
    spriteBatch.Draw(spaceTexture, //sprite
        Vector2.Zero, //position re
window
        null, //rectangle
        Color.White, //tint
        0, //rotation
        spaceOrigin, //origin
        backgroundScale, //scale
        SpriteEffects.None,
        1.0f); //layer, near the
back

    spriteBatch.End();

    // Turn on blending to draw the UFO in the
    // foreground. Note the update for XNA 4.0.
//

```

```

spriteBatch.Begin(SpriteBlendMode.AlphaBlend);
    spriteBatch.Begin(SpriteSortMode.Deferred,
BlendState.AlphaBlend);
        //Center the UFO at the point in the window
        // identified as UFO position. The UFO image
is
        // 64x33 pixels.
        Vector2 origin = new Vector2(32,16);
        spriteBatch.Draw(ufoTexture, //sprite
                        ufoPosition, //position re
window
                        null, //rectangle
                        Color.White, //tint
                        0, //rotation
                        origin, //origin
                        1, //scale
                        SpriteEffects.None,
                        0f); //layer, 0 is in front

        spriteBatch.End();

        base.Draw(gameTime);
    } //end Draw method
} //end class
} //end namespace

```

-end-

Xna0126-Using OOP - A Simple Sprite Class

Learn to design, create, and use a simple Sprite class. Also learn to use a generic List object.

Revised: Mon May 09 13:00:37 CDT 2016

This page is part of a Book titled [XNA Game Studio](#).

Table of Contents

- [Table of Contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [The three pillars of OOP](#)
 - [Wanted, lots of sprites](#)
 - [Encapsulation to the rescue](#)
- [Preview](#)
 - [Demonstrate how to use the Sprite class](#)
 - [Instantiate Sprite objects on the move](#)
 - [Seven Sprite objects](#)
 - [Move to the right and down](#)
- [Discussion and sample code](#)
 - [The Sprite class](#)
 - [Two instance variables and a property accessor method](#)
 - [Two overloaded constructors](#)

- [No image when instantiated](#)
 - [Image loaded during construction](#)
 - [The first parameter](#)
 - [The second parameter](#)
 - [Loading the image](#)
- [The SetImage method](#)
 - [Load an image into a Sprite object](#)
 - [The parameters and the body](#)
- [The Draw method of the Sprite class](#)
 - [A single parameter](#)
 - [The body of the Sprite.Draw method](#)
- [The end of the class](#)
- [The Game1 class](#)
 - [The generic List class](#)
 - [The modified Game1 constructor](#)
 - [The overridden LoadContent method](#)
 - [A new Sprite object](#)
 - [List capacity considerations](#)
 - [Assign an image to the Sprite object](#)
 - [The overridden Update method](#)
 - [The Count property of the List object](#)
 - [The remaining 23 Sprite objects](#)
 - [Adding Sprite object references to the list](#)
 - [Instantiate new Sprite objects](#)
 - [The modulus operator](#)
 - [Every eighth iteration](#)
 - [Even and odd sprite images](#)

- [Make all the existing sprites move](#)
 - [Use a for loop](#)
 - [Put a new value in the Position property.](#)
- [Load a green ball image in the topmost Sprite object](#)
- [Maintain the frame counter](#)
- [The end of the overridden Update method](#)
- [The overridden Game1.Draw method](#)
 - [Draw all Sprite objects](#)
 - [Call the SpriteBatch.Draw method](#)
 - [The end of the Game1.Draw method](#)
- [Run the program](#)
- [Run my_program](#)
- [Summary](#)
- [Miscellaneous](#)
- [Complete program listing](#)

Preface

This module is one in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

An earlier module titled [Getting Started](#) provided information on how to get started programming with Microsoft's XNA Game Studio.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Seven Sprite objects.
- [Figure 2](#). Twenty-four Sprite objects with a green one at the top.

Listings

- [Listing 1](#). Beginning of the Sprite class.
- [Listing 2](#). Two overloaded constructors.
- [Listing 3](#). The SetImage method.
- [Listing 4](#). The Draw method of the Sprite class.
- [Listing 5](#). Beginning of the class named Game1.
- [Listing 6](#). The modified constructor for the Game1 class.
- [Listing 7](#). The overridden LoadContent method.
- [Listing 8](#). Beginning of the overridden Update method.
- [Listing 9](#). Instantiate new Sprite objects.
- [Listing 10](#). Make all the existing sprites move.
- [Listing 11](#). Load a green ball image in the topmost Sprite object.
- [Listing 12](#). Maintain the frame counter.
- [Listing 13](#). The overridden Game1.Draw method.
- [Listing 14](#). Contents of the file named Sprite.cs
- [Listing 15](#). Contents of the file named Game1.cs.

General background information

The three pillars of OOP

An object-oriented programming language like C# supports *encapsulation* , *inheritance* , and *polymorphism* . In this module, you will learn how to take advantage of encapsulation.

Wanted, lots of sprites

Assume that you are writing a game program in which you need to have several dozen similar sprites on the screen at the same time. Creating and controlling that many sprites without using encapsulation would require you to write a lot of program code.

Encapsulation to the rescue

However, by encapsulating the characteristics of a sprite into a class, which is a blueprint for an object, you can instantiate sprite objects just like cutting out cookies with a cookie cutter.

In this module, you will learn to define and use a very simple **Sprite** class. In future modules, you will learn how to modify the **Sprite** class to make it more sophisticated by making use of inheritance and polymorphism in addition to encapsulation

Preview

The XNA project that I will explain in this module is named **XNA0126Proj**. This project demonstrates how to design and use a very simple version of a **Sprite** class. An object instantiated from the **Sprite** class has the following general characteristics:

- It can be instantiated without an image.
- It can be instantiated with an image.
- It can have its image set.
- It can have its position within the game window set.
- It can cause itself to be drawn.

Demonstrate how to use the Sprite class

Methods are overridden in the standard XNA **Game1** class that demonstrate the use of the **Sprite** class.

One **Sprite** object is instantiated in the overridden **LoadContent** method of the **Game1** class. The object's reference is saved in a generic **List** object. Twenty-three more **Sprite** objects are instantiated in the overridden **Update** method while the game loop is running.

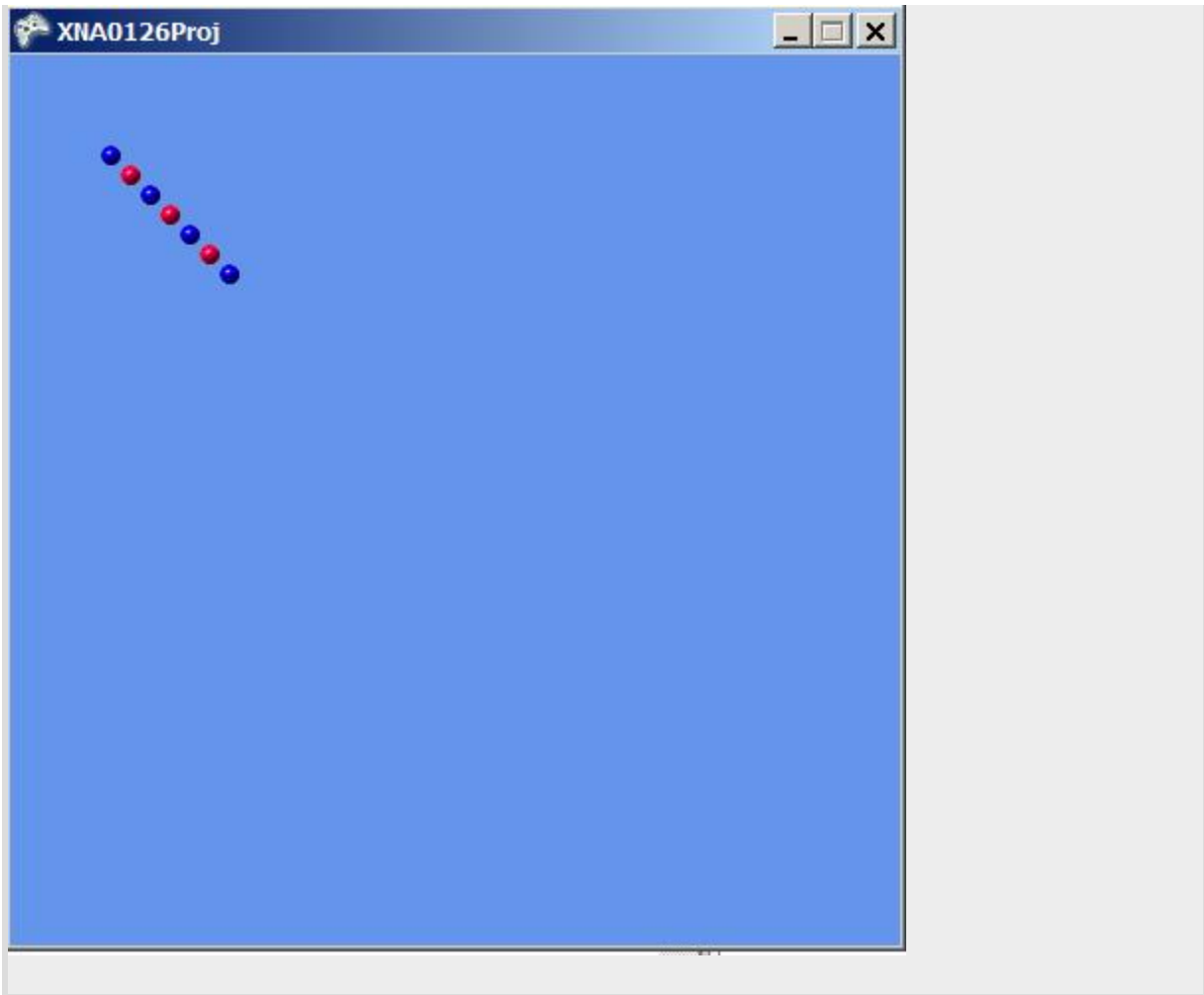
Instantiate Sprite objects on the move

A new **Sprite** object is instantiated in the **Update** method every 8th iteration of the game loop until **twenty-four Sprite** objects have been instantiated. The object's references are saved in the same generic **List** object mentioned above.

An image of a blue ball is stored in 12 of the objects and an image of a red ball is stored in the other 12 objects. The red and blue balls alternate and the **Sprite** objects are drawn in a diagonal line as shown in [Figure 1](#).

Note:

Figure 1 . Seven Sprite objects.



Seven Sprite objects

[Figure 1](#) shows the game window after seven of the twenty-four **Sprite** objects have been instantiated and drawn in the game window.

Move to the right and down

The line of **Sprite** objects moves across the game window from upper left to lower right as the **Sprite** objects are being instantiated. They stop moving when they reach the bottom right corner of the game window.

When the objects stop moving, the image in the topmost **Sprite** object is changed from a blue ball to a green ball as shown in [Figure 2](#).

Note:

Figure 2 . Twenty-four Sprite objects with a green one at the top.



Discussion and sample code

As usual, I will explain the program code in fragments. A complete listing of the class named **Sprite** is provided in [Listing 14](#) and a complete listing of the class named **Game1** is provided in [Listing 15](#).

I will begin my explanation with the class named **Sprite** .

The Sprite class

The file named **Sprite.cs** (see [Listing 14](#)) defines a simple version of a **Sprite** class from which multiple **Sprite** objects can be instantiated, loaded with an image, and drawn in the game window. The **Position** property of each **Sprite** object can be accessed by the user to control the position at which the sprite is drawn.

The definition of the **Sprite** class begins in [Listing 1](#).

Note:

Listing 1 . Beginning of the Sprite class.

```
namespace XNA0126Proj {
    class Sprite {
        private Texture2D texture;
        private Vector2 position = new Vector2(0,0);
        //-----
        -----//

        public Vector2 Position {
            get {
                return position;
            } //end get
            set {
                position = value;
            } //end set
        } //end Position property accessor
    }
}
```

Two instance variables and a property accessor method

[Listing 1](#) declares two instance variables and defines an accessor for the **Position** property.

The first instance variable named **texture** will be used to store the image for the sprite. The second instance variable named **position** will be used to store the value for the **Position** property.

Two overloaded constructors

[Listing 2](#) defines two overloaded constructors for the **Sprite** class.

Note:

Listing 2 . Two overloaded constructors.

```
public Sprite() { //constructor
} //end noarg constructor
//-----
-----//

public Sprite(String assetName,
               ContentManager contentManager) {
    texture =
        contentManager.Load<Texture2D>
(assetName);
} //end constructor
```

No image when instantiated

The first overloaded constructor, which requires no parameters, makes it possible to instantiate a **Sprite** object without loading an image for the object when it is constructed. A method named **SetImage** can be called later to load an image for the object.

Note: Be aware that if you instantiate a **Sprite** object using this constructor and then attempt to draw the object without first calling the **SetImage** method to load an image into the object, the program will fail with a runtime error.

Image loaded during construction

The second overloaded constructor, which requires two parameters, makes it possible to load an image for the **Sprite** object when it is constructed.

The first parameter

The first parameter required by the second constructor is the **Asset Name** property for an image file that is added to the **Content** folder during the project design phase.

The second parameter

The second parameter is a reference to the **ContentManager** object that is inherited into the **Game1** object from the **Game** class.

Loading the image

You are already familiar with the code in the body of the constructor that is used to load the image into the object.

The SetImage method

The **SetImage** method is shown in its entirety in [Listing 3](#).

Note:

Listing 3 . The SetImage method.

```
public void SetImage(String assetName,
                    ContentManager
contentManager) {
    texture =
        contentManager.Load<Texture2D>
(assetName);
} //end SetImage
```

Load an image into a Sprite object

The **SetImage** method makes it possible to load an image into a **Sprite** object that was originally [constructed without an image](#), or to change the image in a **Sprite** object that already contains an image.

The parameters and the body

The parameters required by this method and the body of the method are the same as for the [first constructor](#) discussed above.

The Draw method of the Sprite class

[Listing 4](#) shows the **Draw** method of the **Sprite** class in its entirety.

Note:

Listing 4 . The Draw method of the Sprite class.

```
public void Draw(SpriteBatch spriteBatch) {
```

```
spriteBatch.Draw(texture, position, Color.White);  
    }//end Draw method  
    //-----  
-----//  
    }//end Sprite class  
}//end namespace
```

This method should be called after a call to the **SpriteBatch.Begin** method and before a call to the **SpriteBatch.End** method.

A single parameter

The method requires a single parameter, which is a reference to the **SpriteBatch** object on which the **Begin** method has been called.

The body of the Sprite.Draw method

You should recognize the single statement in the method as a call to the simplest available **Draw** method belonging to the **SpriteBatch** object. This version of the **SpriteBatch.Draw** method allows the caller to specify

- The texture or image to be drawn.
- The position in the game window relative to the upper left corner of the window at which the image will be drawn.
- A color tint that will be applied to the image with White being no change in color.

The end of the class

[Listing 4](#) signals the end of the **Sprite** class.

The Game1 class

Methods of the **Game1** class were overridden to demonstrate the use of the **Sprite** class to produce the output described [earlier](#).

The class named **Game1** begins in [Listing 5](#).

Note:

Listing 5 . Beginning of the class named Game1.

```
namespace XNA0126Proj {

    public class Game1 :
Microsoft.Xna.Framework.Game {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        //References to the Sprite objects are stored
in this
        // List object.
        List<Sprite> sprites = new List<Sprite>();

        int maxSprites = 24; //Max number of sprites.
        int frameCnt = 0; //Game loop frame counter

        //This is the limit on the number of frames in
which
        // the sprites are moved.
        int moveLim = 200;
```

[Listing 5](#) simply declares several instance variables, most of which you should recognize. The others are well described by the comments. However,

one of the instance variables, **sprites** , introduces a concept that is new to this module.

The generic List class

[Listing 5](#) declares a variable named **sprites** and populates it with a reference to a new generic **List** object that is conditioned to store and retrieve references to objects of the class **Sprite** .

Here is some of what the [documentation](#) has to say about the generic **List** class:

Note: "Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists."

The generic **List** class provides many more capabilities than I will use in this program. I will use an object of the generic **List** class to store references to **Sprite** objects that I can later access using a zero-based index. This will eliminate the requirement to declare a separate reference variable for each of the **Sprite** objects that I instantiate.

The modified Game1 constructor

The constructor for the **Game1** class was modified to set the size of the game window as shown in [Listing 6](#) .

Note:
Listing 6 . The modified constructor for the Game1 class.

```
public Game1() { //constructor
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";

    //Set the size of the game window.
    graphics.PreferredBackBufferWidth = 450;
    graphics.PreferredBackBufferHeight = 450;
} //end constructor
```

You have seen code identical to this code in earlier modules so there is nothing new to discuss here.

The overridden LoadContent method

The overridden **LoadContent** method is shown in [Listing 7](#).

Note:

Listing 7 . The overridden LoadContent method.

```
protected override void LoadContent() {
    spriteBatch = new
SpriteBatch(GraphicsDevice);

    //Create the first sprite in the LoadContent
    // method using the noarg constructor.
    sprites.Add(new Sprite());

    //Assign an image to the sprite.
    sprites[0].SetImage("blueball",Content);

} //end LoadContent
```


A new Sprite object

A statement near the center of [Listing 7](#) instantiates a new **Sprite** object with no image and calls the **Add** method of the generic **List** object to add the object's reference to the end of the list.

List capacity considerations

One advantage of using a generic **List** object as an alternative to a simple array is that it is not necessary to declare the capacity of the **List** object when the program is compiled. The capacity of the **List** object increases automatically as necessary to accommodate all of the references that are added to the list.

Since this is the first reference added to the list, it can be accessed later using an index value of 0.

Assign an image to the Sprite object

The last statement in [Listing 7](#) retrieves the reference from the list at index 0 and uses that reference to call the **SetImage** method on the **Sprite** object to which it refers.

You learned about the **SetImage** method in the discussion of the code in [Listing 3](#). This call causes the **Sprite** object whose reference is stored at index 0 in the list to load the image from the image file with the **Asset Name** property value of "blueball".

As mentioned earlier, the second parameter named **Content** is a reference to a **ContentManager** object that is inherited from the **Game** class.

The overridden Update method

As you learned [earlier](#), this program creates, manipulates, and draws 24 objects of the **Sprite** class in the game window as shown in [Figure 2](#). The

first **Sprite** object was created in the overridden **LoadContent** method when it was called earlier.

The Count property of the List object

The generic **List** object referred to by **sprites** has a property named **Count** that keeps track of the number of references contained in the object. The first time the **Update** method is called, the value of **Count** is 1 as a result of the **Sprite** object's reference having been added to the list in the **LoadContent** method earlier.

The remaining 23 Sprite objects

This program creates the remaining 23 sprites in the **Update** method to simulate a game in which sprites come and go as the game progresses.

The overridden **Update** method begins in [Listing 8](#).

Note:

Listing 8 . Beginning of the overridden Update method.

```
protected override void Update(GameTime
gameTime) {

    if(sprites.Count < (maxSprites)) {
```

Adding Sprite object references to the list

[Listing 8](#) shows the beginning of an **if** statement in which a new **Sprite** objects reference will be added to the list every eighth iteration (*frame*) of the game loop until all 24 sprites have been added.

Note that a frame counter named **frameCnt** is declared and initialized to zero in [Listing 5](#) and is incremented near the end of the overridden **Update** method in [Listing 12](#).

Instantiate new Sprite objects

The code in [Listing 9](#) uses the modulus (%) operator to identify every eighth iteration of the game loop and to instantiate a new **Sprite** object during those iterations.

Note:

Listing 9 . Instantiate new Sprite objects.

```
        if(frameCnt % 8 == 0) {
            //Instantiate a new sprite every 8th
frame.
            if((sprites.Count) % 2 == 0) {
                //Even numbered sprites
                sprites.Add(new
Sprite("blueball",Content));
            }
            else {
                //Odd numbered sprites
                sprites.Add(new
Sprite("redball",Content));
            } //end else
        } //end if on frameCnt
    } //end if on sprites.Count
```

The modulus operator

In case you have forgotten, the modulus operator returns the remainder of a division instead of returning the quotient. If an integer value is divided by 8, the returned value is 0 only when the integer value is a multiple of 8. (Also by definition, $0 \% 8$ returns 0.)

Every eighth iteration

Therefore, the conditional expression in the first **if** statement in [Listing 9](#) will allow the statements contained in the body of the **if** statement, (which instantiate new **Sprite** objects), to be executed only during every eighth iteration of the game loop.

Further, the conditional expression in [Listing 8](#) will not allow the code in [Listing 9](#) to be executed after 24 **Sprite** objects have been instantiated.

Even and odd sprite images

The conditional expression in the second **if** statement in [Listing 9](#) causes the new **Sprite** objects that are instantiated to alternate between the "blueball" and "redball" images shown in [Figure 1](#).

Make all the existing sprites move

[Listing 5](#) declares an instance variable named **moveLim** and sets its value to 200. The code in [Listing 10](#) causes all of the existing sprites to move to the right and down if the value of the frame counter is less than 200.

Note:

Listing 10 . Make all the existing sprites move.

```
if(frameCnt < moveLim) {  
    for(int cnt = 0; cnt < sprites.Count; cnt++)
```

```
{  
    sprites[cnt].Position = new Vector2(  
        10 * cnt + frameCnt, 10 * cnt +  
frameCnt);  
    }//end for loop  
}//end if
```

Use a for loop

The code in [Listing 10](#) uses a **for** loop to access each of the **Sprite** object references currently stored in the list, iterating from 0 to one less than the count of references stored in the list given by **sprites.Count** .

Put a new value in the Position property

Once a **Sprite** object's reference has been accessed, [Listing 10](#) sets the **Position** property stored in the object to a new **Vector2** object for which the **X** and **Y** values have been modified on the basis of the frame counter.

The new **X** and **Y** values cause the object to be drawn a little further down and to the right the next time it is drawn relative to its current position. This causes the entire diagonal line of **Sprite** objects to move down and to the right in the game window.

Load a green ball image in the topmost Sprite object

[Listing 11](#) calls the **SetImage** method to change the image in the topmost sprite at the end of the run from a blue ball to a green ball.

Note:

Listing 11 . Load a green ball image in the topmost Sprite object.

```
if(frameCnt == moveLim) {  
    sprites[0].SetImage("greenball",Content);  
}//end if
```

This capability would be useful, for example to change a sprite's image into a fireball in the event of a collision with another sprite.

Maintain the frame counter

The code in [Listing 12](#) keeps track of the count of the first **moveLim** iterations of the game loop.

Note:

Listing 12 . Maintain the frame counter.

```
if(frameCnt < moveLim) {  
    frameCnt++;  
}//end if  
  
base.Update(gameTime);  
}//end Update method
```

The end of the overridden Update method

Then [Listing 12](#) makes the required call to the **Update** method in the superclass and signals the end of the method.

The overridden Game1.Draw method

The overridden **Game1.Draw** method is shown in its entirety in [Listing 13](#).

Note:

Listing 13 . The overridden Game1.Draw method.

```
        protected override void Draw(GameTime
gameTime) {
            GraphicsDevice.Clear(Color.CornflowerBlue);

            spriteBatch.Begin();

            //Draw all sprites.
            for(int cnt = 0;cnt < sprites.Count;cnt++) {
                sprites[cnt].Draw(spriteBatch);
            }//end for loop

            spriteBatch.End();

            base.Draw(gameTime);
        }//end Draw method
        //-----
-----//
    }//end class
} //end namespace
```

Draw all Sprite objects

After calling the **SpriteBatch.Begin** method and before calling the **SpriteBatch.End** method, [Listing 13](#) uses a **for** loop to call the **Sprite.Draw** method on every **Sprite** object whose reference is stored in the list, passing a reference to the **SpriteBatch** object as a parameter in each call.

Call the `SpriteBatch.Draw` method

This causes each object to execute the single statement belonging to the **Draw** method shown in [Listing 4](#). Thus, the code in [Listing 13](#) causes each **Sprite** object to call the **SpriteBatch.Draw** method to draw itself at the position specified by the current value of its **Position** property.

Note that there are three different methods named **Draw** being used here:

- `Game1.Draw`
- `SpriteBatch.Draw`
- `Sprite.Draw`

The end of the `Game1.Draw` method

After calling the **SpriteBatch.End** method, [Listing 13](#) makes the required call to the superclass' **Game.Draw** method and then signals the end of the **Game1.Draw** method. [Listing 13](#) also signals the end of the class and the end of the program.

Run the program

I encourage you to copy the code from [Listing 14](#) and [Listing 15](#). Use that code to create an XNA project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Run my program

Click [here](#) to download a zip file containing my version of the program. Extract the folder named **XNA0126Proj** from the zip file and save it somewhere on your disk. Start **Visual C# 2010 Express** and select **Open Project...** from the **File** menu. Navigate to the project folder and select the file with the extension of **.sln**. This should cause the project to open and be ready to run or debug as described in the earlier module titled [Getting Started](#).

Summary

You learned how to design, create, and use a simple **Sprite** class. You also learned how to use a generic **List** object.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Xna0126-Using OOP - A Simple Sprite Class
- File: Xna0126.htm
- Published: 02/28/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

Complete listings of the XNA program files discussed in this module are provided in [Listing 14](#) and [Listing 15](#) below.

Note:

Listing 14 . Contents of the file named Sprite.cs

```
/*Project XNA0126Proj
 * This file defines a very simple version of a
Sprite
 * class from which multiple Sprite objects can be
 * instantiated, loaded with an image, and drawn.
 * The Position property can be accessed by the
user
 * to control the position at which the sprite is
drawn.

*****
*****/

using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;

namespace XNA0126Proj {
    class Sprite {
        private Texture2D texture;
        private Vector2 position = new Vector2(0,0);
        //-----
        -----//

        public Vector2 Position {
            get {
```



```

        texture =
            contentManager.Load<Texture2D>
(assetName);
    }//end SetImage
    //-----
-----//

    public void Draw(SpriteBatch spriteBatch) {
        //Call the simplest available version of
        // spriteBatch.Draw

spriteBatch.Draw(texture, position, Color.White);
    }//end Draw method
    //-----
-----//
    }//end class
}//end namespace

```

Note:

Listing 15 . Contents of the file named Game1.cs.

```

/*Project XNA0126Proj
 * This project demonstrates how to design and use
a very
 * simple version of a Sprite class.
 *
 * One Sprite object is instantiated in the
LoadContent
 * method. The object's reference is saved in a
generic
 * List object.

 *
 * Twenty-three more Sprite objects are
instantiated

```

```

* while the game loop is running. A new object is
* instantiated every 8th iteration of the game
loop
* until 24 objects have been instantiated. Their
* references are saved in a generic List object.
*
* An image of a blueball is stored in 12 of the
objects
* and an image of a redball is stored in the
other 12
* objects.
*
* The Sprite objects are drawn in a diagonal line
in
* the game window. The line of Sprite objects
moves
* across the game window from upper left to lower
right.
* The Sprite objects stop moving when they reach
the
* bottom right corner of the game window.
*
* When the objects stop moving, the image in the
* topmost Sprite object is changed from a
blueball to a
* greenball.
*

```

```

*****

```

```

*** /

```

```

using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using XNA0126Proj;

```

```

namespace XNA0126Proj {

```

```

    public class Game1 :
Microsoft.Xna.Framework.Game {
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    //References to the Sprite objects are stored
in this
    // List object.
    List<Sprite> sprites = new List<Sprite>();

    int maxSprites = 24;//Max number of sprites.
    int frameCnt = 0;//Game loop frame counter

    //This is the limit on the number of frames in
which
    // the sprites are moved.
    int moveLim = 200;
    //-----
-----//

    public Game1() {//constructor
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        //Set the size of the game window.
        graphics.PreferredBackBufferWidth = 450;
        graphics.PreferredBackBufferHeight = 450;
    }//end constructor
    //-----
-----//

    protected override void Initialize() {
        //No initialization required.
        base.Initialize();
    }//end Initialize
    //-----
-----//

```

```

        protected override void LoadContent() {
            spriteBatch = new
SpriteBatch(GraphicsDevice);

            //Create the first sprite in the LoadContent
            // method using the noarg constructor.
            sprites.Add(new Sprite());
            //Assign an image to the sprite.
            sprites[0].SetImage("blueball",Content);
            //More content is loaded in the Update
method.
        }//end LoadContent
        //-----
        -----//

        protected override void UnloadContent() {
            //No content unload required.
        }//end unloadContent
        //-----
        -----//

        protected override void Update(GameTime
gameTime) {

            //Create remaining sprites in the Update
method to
            // simulate a game in which sprites come and
go as
            // the game progresses.
            if(sprites.Count < (maxSprites)) {
                if(frameCnt % 8 == 0) {
                    //Instantiate a new sprite every 8th
frame.

                    if((sprites.Count) % 2 == 0) {
                        //Even numbered sprites
                        sprites.Add(new

```

```

Sprite("blueball",Content));
    }
    else {
        //Odd numbered sprites
        sprites.Add(new
Sprite("redball",Content));
    }//end else
} //end if on frameCnt
} //end if on sprites.Count

//Make all the sprites move.
if(frameCnt < moveLim) {
    for(int cnt = 0;cnt < sprites.Count;cnt++)
{
    sprites[cnt].Position = new Vector2(
        10 * cnt + frameCnt,10 * cnt +
frameCnt);
    } //end for loop
} //end if

//Change the image on the first sprite at
the end
// of the run. Could be used, for example to
// change a sprite's image to a fireball in
the
// event of a collision.
if(frameCnt == moveLim) {
    sprites[0].SetImage("greenball",Content);
} //end if

//Keep track of the count of the first
moveLim
// iterations of the game loop.
if(frameCnt < moveLim) {
    frameCnt++;
} //end if

```



```
        base.Update(gameTime);
    }//end Update method
    //-----
-----//

    protected override void Draw(GameTime
gameTime) {
        GraphicsDevice.Clear(Color.CornflowerBlue);

        spriteBatch.Begin();

        //Draw all sprites.
        for(int cnt = 0;cnt < sprites.Count;cnt++) {
            sprites[cnt].Draw(spriteBatch);
        }//end for loop

        spriteBatch.End();

        base.Draw(gameTime);
    }//end Draw method
    //-----
-----//
    }//end class
} //end namespace
```

-end-

Xna0128-Improving the Sprite Class

Learn how to make improvements to the Sprite class that was introduced in an earlier lesson.

Revised: Mon May 09 17:04:23 CDT 2016

This page is part of a Book titled [XNA Game Studio](#).

Table of Contents

- [Table of Contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Preview](#)
 - [The screen output](#)
 - [Lots and lots of sprites](#)
- [Discussion and sample code](#)
 - [The Sprite class](#)
 - [Properties](#)
 - [Other instance variables](#)
 - [The first three property accessor methods](#)
 - [The property accessor method for Speed](#)
 - [The set side](#)
 - [The get side](#)
 - [The constructor](#)

- [Load an image](#)
 - [A random number generator](#)
 - [The SetImage method](#)
 - [The Move method](#)
 - [To move or not to move](#)
 - [Keeping up on the average](#)
 - [Code in the body of the if statement](#)
 - [Add the direction vector to the position vector](#)
 - [A collision with an edge of the game window](#)
 - [Modify the position and call the NewDirection method](#)
 - [The end of the Move method](#)
 - [The method named NewDirection](#)
 - [The length of the direction vector and the signs of the components](#)
 - [Compute the hypotenuse](#)
 - [Use the conditional operator](#)
 - [Compute components of a new direction vector](#)
 - [Compute a new random value for the X component](#)
 - [Compute a consistent value for the Y component](#)
 - [Adjust the signs of the X and Y components](#)
 - [A new direction vector with the same length in the same quadrant](#)
 - [The Draw method](#)
 - [The end of the Sprite class](#)
- [The Game1 class](#)

- [Instance variables](#)
- [The modified constructor](#)
- [The overridden LoadContent method](#)
 - [Instantiate all of the space rock sprites](#)
 - [Call the Sprite constructor](#)
 - [Set the property values](#)
 - [Instantiate and set properties on the power pills and the UFOs](#)
- [The private DirectionVector method](#)
 - [Return a direction vector](#)
 - [The signs of the components](#)
- [The overridden Update method](#)
 - [Very simple code](#)
 - [To move or not to move, that is the question](#)
 - [A characteristic of an object-oriented program](#)
- [The overridden Game.Draw method](#)
 - [Erase and redraw the entire game window](#)
- [Not the same approach as some other game engines](#)
- [The end of the program](#)
- [Run the program](#)
- [Run my program](#)
- [Summary](#)
- [What's next?](#)
- [Miscellaneous](#)
- [Complete program listing](#)

Preface

This module is one in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

An earlier module titled [Getting Started](#) provided information on how to get started programming with Microsoft's XNA Game Studio.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Screen shot of the running program.

Listings

- [Listing 1](#). Beginning of the Sprite class.
- [Listing 2](#). The first three property accessor methods.
- [Listing 3](#). The property accessor method for Speed.
- [Listing 4](#). The constructor for the Sprite class.
- [Listing 5](#). The SetImage method.
- [Listing 6](#). Beginning of the Move method.
- [Listing 7](#). Add the direction vector to the position vector.
- [Listing 8](#). Process a collision with an edge of the game window.
- [Listing 9](#). Beginning of the method named NewDirection.
- [Listing 10](#). Compute components of a new direction vector.
- [Listing 11](#). The Sprite.Draw method.
- [Listing 12](#). Beginning of the Game1 class.
- [Listing 13](#). The modified constructor.

- [Listing 14](#). Beginning of the overridden LoadContent method.
- [Listing 15](#). Instantiate and set properties on the power pills and the UFOs.
- [Listing 16](#). The private DirectionVector method.
- [Listing 17](#). Tell the sprites to move.
- [Listing 18](#). The overridden Game.Draw method.
- [Listing 19](#). The class named Sprite for the project named XNA0128Proj.
- [Listing 20](#). The class named Game1 for the project named XNA0128Proj.

General background information

You learned how to design, create, and use a simple **Sprite** class in an earlier module. You also learned to use a generic **List** object to store references to objects of the **Sprite** class.

Preview

I will explain improvements made to the **Sprite** class and will show you how to write a **Game1** class that takes advantage of those improvements.

The screen output

[Figure 1](#) shows a reduced screen shot of the program while it is running.

Note:

Figure 1 . Screen shot of the running program.



Lots and lots of sprites

When this screen shot was taken, the program had 24 space rocks, 12 red power pills and six UFOs all navigating in the game window.

Discussion and sample code

As usual, I will discuss and explain the program code in fragments. A complete listing of the **Sprite** class is provided in [Listing 19](#) and a complete listing of the **Game1** class is provided in [Listing 20](#).

The Sprite class

The **Sprite** class begins in [Listing 1](#) with the declaration of several instance variables.

Note:

Listing 1 . Beginning of the Sprite class.

```
namespace XNA0128Proj {
    class Sprite {
        private Texture2D image;
        private Vector2 position = new Vector2(0,0);
        private Vector2 direction = new Vector2(0,0);
        private Point windowSize;
        private Random random;
        double elapsedTime;//in milliseconds
        //The following value is the inverse of speed
in
        // moves/msec expressed in msec/move.
        double elapsedTimeTarget;
```

The purpose of these instance variables will become clear later in the discussion.

Properties

This class has the following properties:

- **Position** - position of the sprite in the game window relative to the upper left corner of the game window. The X and Y values are saved as type **Vector2** in the instance variable named **position** .
- **WindowSize** - The dimensions of the game window. The width and height values are saved as type **Point** in the instance variable named **windowSize** .
- **Direction** - Direction of motion of the sprite expressed as a 2D vector with an X and a Y component. The length of this vector is the distance that the sprite moves each time it moves. The values are saved as type **Vector2** in the instance variable named **direction** .

- **Speed** - Speed of the sprite expressed in moves per millisecond. The actual speed in pixels per millisecond is the product of this value and the length of the direction vector.

Other instance variables

In addition, the class has the following instance variables that are set either by the constructor or by a method:

- **image** - the image that is drawn to represent the sprite in the game window. The image can be modified as the game progresses through calls to the **SetImage** method.
- **random** - a reference to a random number generator.
- **elapsedTime** - used to keep track of the amount of elapsed time in milliseconds since the last time that the sprite was actually moved.
- **elapsedTimeTarget** - will be explained later.

The first three property accessor methods

The first three property accessor methods are shown in [Listing 2](#).

Note:

Listing 2 . The first three property accessor methods.

```
//Position property accessor
public Vector2 Position {
    get {
        return position;
    }
    set {
        position = value;
    } //end set
} //end Position property accessor
```

```

//-----
-----//

//WindowSize property accessor
public Point WindowSize {
    set {
        windowSize = value;
    }//end set
}//end WindowSize property accessor
//-----
-----//

//Direction property accessor
public Vector2 Direction {
    get {
        return direction;
    }
    set {
        direction = value;
    }//end set
}//end Direction property accessor

```

These three accessor methods are straightforward and shouldn't require further explanation.

The property accessor method for Speed

Humans usually find it easier to think in terms of speed such as *miles per hour* while it is sometimes easier to write computer programs that deal with the reciprocal of speed such as *hours per mile* .

The property accessor method for the property named **Speed** is shown in [Listing 3](#).

Note:

Listing 3 . The property accessor method for Speed.

```
//Speed property accessor. The set side should
be
// called with speed in moves/msec. The get
side
// returns speed moves/msec.
public double Speed {
    get {
        //Convert from elapsed time in msec/move
to
        // speed in moves/msec.
        return elapsedTimeTarget/1000;
    }
    set {
        //Convert from speed in moves/msec to
        // elapsed time in milliseconds/move.
        elapsedTimeTarget = 1000/value;
    } //end set
} //end Speed property accessor
```

The set side

The **set** side of the property accessor method for the **Speed** property receives the incoming value as *moves per millisecond* . The code converts this value to *milliseconds per move* and saves it in the instance variable named **elapsedTimeTarget** mentioned earlier.

This is the target for the elapsed time in milliseconds from one movement to the next movement of the sprite in the game window. Every time a sprite moves, it moves the same distance. Therefore, the apparent speed of sprite movement as seen by the viewer can be controlled by controlling the elapsed time between movements.

The get side

The **get** side of the property accessor method for the **Speed** property converts the returned value from *milliseconds per move* back to *moves per millisecond* .

The constructor

The constructor for the Sprite class is shown in [Listing 4](#).

Note:

Listing 4 . The constructor for the Sprite class.

```
public Sprite(String assetName,  
               ContentManager contentManager,  
               Random random) {  
    image = contentManager.Load<Texture2D>  
(assetName);  
    this.random = random;  
} //end constructor
```

Load an image

The constructor loads an image for the sprite when it is instantiated. Therefore, it requires an **Asset Name** for the image and a reference to a **ContentManager** object.

A random number generator

The constructor also requires a reference to a **Random** object capable of generating a sequence of pseudo random values of type **double** .

Note: The purpose of the random number generator will become clear later.

The program should use the same **Random** object for all sprites to avoid getting the same sequence of values for different sprites when two or more sprites are instantiated in a very short period of time.

The SetImage method

The **SetImage** method is shown in [Listing 5](#).

Note:

Listing 5 . The SetImage method.

```
public void SetImage(String assetName,
                    ContentManager
contentManager) {
    image = contentManager.Load<Texture2D>
(assetName);
} //end SetImage
```

This method can be called to load a new image for an existing sprite. The method is essentially the same as a method having the same name that I explained in an earlier module, so no further explanation should be required.

The Move method

This method causes the sprite to move in the direction of the direction vector if the elapsed time since the last move exceeds the elapsed time target based on the specified speed.

The **Move** method begins in [Listing 6](#).

Note:

Listing 6 . Beginning of the Move method.

```
public void Move(GameTime gameTime) {  
    //Accumulate elapsed time since the last  
move.  
    elapsedTime +=  
gameTime.ElapsedGameTime.Milliseconds;  
  
    if(elapsedTime > elapsedTimeTarget){  
        //It's time to make a move. Set the  
elapsed  
        // time to a value that will attempt to  
produce  
        // the specified speed on the average.  
        elapsedTime -= elapsedTimeTarget;  
    }  
}
```

The sprite doesn't necessarily move every time the **Move** method is called. Instead, it uses the incoming parameter to compute the elapsed time since the last time that it actually moved.

To move or not to move

If that elapsed time exceeds the target that is based on the specified speed in *moves/millisecond* , then it reduces the elapsed time value by the target

value and makes an adjustment to the **position** value. Changing the **position** value will cause the sprite to move in the game window the next time it is drawn.

Keeping up on the average

By reducing the elapsed time by the target time instead of setting it to zero, the sprite attempts to achieve the target speed *on the average* . For example, assume that for some reason, there is a long delay between calls to the **Move** method and the elapsed time value is two or three times greater than the target time. This means that the sprite has gotten behind and is not in the position that it should be in. In that case, the sprite will move every time the **Move** method is called for several successive calls to the **Move** method. (In other words, the sprite will experience a short spurt in speed.) This should cause it to catch up and be in the correct position once it does catch up.

Of course, if the elapsed time between calls to the **Move** method is greater than the target time over the long term, the sprite will never be able to keep up.

Code in the body of the if statement

If the conditional expression for the **if** statement in [Listing 6](#) returns true, then the last statement in [Listing 6](#) along with the remainder of the body of the **if** statement will be executed. Otherwise, that statement and the remaining body of the **if** statement will be skipped.

The remaining body of the **if** statement begins in [Listing 7](#).

Add the direction vector to the position vector

One of the advantages of treating the position and the direction as 2D vectors based on the structure named [Vector2](#) is that the **Vector2** structure provides various [methods](#) that can be used to manipulate vectors.

The code in [Listing 7](#) calls the [Add](#) method of the **Vector2** class to add the direction vector to the position vector returning the sum of the two vectors. The sum is saved as the new position vector.

Note:

Listing 7 . Add the direction vector to the position vector.

```
position =  
Vector2.Add(position,direction);
```

In case you are unfamiliar with the addition of 2D vectors, if you add a pair of 2D vectors, the X component of the sum is the sum of the X components and the Y component of the sum is the sum of the Y components.

A collision with an edge of the game window

The code in [Listing 8](#) checks for a collision with an edge of the game window.

If the sprite collides with an edge, the code in [Listing 8](#) causes the sprite to wrap around and reappear at the opposite edge, moving at the same speed in a different direction within the same quadrant as before. In other words, if a sprite is moving down and to the right and collides with the right edge of the window, it will reappear at the left edge, still moving down and to the right but not in exactly the same direction down and to the right.

Note:

Listing 8 . Process a collision with an edge of the game window.


```

        if(position.X < -image.Width){
            position.X = windowSize.X;
            NewDirection();
        }//end if

        if(position.X > windowSize.X){
            position.X = -image.Width/2;
            NewDirection();
        }//end if

        if(position.Y < -image.Height) {
            position.Y = windowSize.Y;
            NewDirection();
        }//end if

        if(position.Y > windowSize.Y){
            position.Y = -image.Height / 2;
            NewDirection();
        }//end if on position.Y
    }//end if on elapsed time
} //end Move

```

Modify the position and call the `NewDirection` method

In all cases shown in [Listing 8](#), if a collision occurs, the position of the sprite is modified to position the sprite at the opposite edge. Then the method named **NewDirection** is called to modify the direction pointed to by the direction vector.

Note: The `NewDirection` method is declared *private* to prevent it from being accessible to code outside the `Sprite` class because it has no meaning outside the `Sprite` class.

The end of the Move method

[Listing 8](#) signals the end of the **Move** method.

The method named NewDirection

The method named **NewDirection** begins in [Listing 9](#).

Note:

Listing 9 . Beginning of the method named NewDirection.

```
private void NewDirection() {  
    double length = Math.Sqrt(  
        direction.X *  
direction.X +  
        direction.Y *  
direction.Y);  
  
    Boolean xNegative = (direction.X < 0)?  
true:false;  
    Boolean yNegative = (direction.Y < 0)?  
true:false;
```

The length of the direction vector and the signs of the components

[Listing 9](#) begins by determining the length of the current direction vector along with the signs of the X and Y components of the vector.

Compute the hypotenuse

The first statement in the method in [Listing 9](#) calls the [Math.Sqrt](#) method and uses the [Pythagorean Theorem](#) to compute the length of the hypotenuse

of the right triangle formed by the X and Y components of the direction vector. This is the **length** of the direction vector.

Use the conditional operator

Then the last two statements in [Listing 9](#) use the [conditional operator](#) to determine if the signs of the components are negative. If so, the variables named **xNegative** and/or **yNegative** are set to true.

Compute components of a new direction vector

Having accomplished that task, the code in [Listing 10](#) computes the components for a new direction vector of the *same length* with the X and Y components having random (but consistent) lengths and the same signs as before.

Compute a new random value for the X component

For the code in [Listing 10](#) to make any sense at all, you must know that the call to **random . NextDouble** returns a pseudo-random value, uniformly distributed between 0.0 and 1.0.

Note:

Listing 10 . Compute components of a new direction vector.

```
        //Compute a new X component as a random
portion of
        // the vector length.
        direction.X =
            (float)(length *
random.NextDouble());
```

```

        //Compute a corresponding Y component that
will
        // keep the same vector length.
        direction.Y = (float)Math.Sqrt(length*length
-
direction.X*direction.X);

        //Set the signs on the X and Y components to
match
        // the signs from the original direction
vector.
        if(xNegative)
            direction.X = -direction.X;
        if(yNegative)
            direction.Y = -direction.Y;
    }//end NewDirection

```

The first statement in [Listing 10](#) computes a new value for the X component of the current direction vector, which is a random portion of the **length** of the current direction vector ranging from 0 to the full length of the vector.

Compute a consistent value for the Y component

Then the second statement in [Listing 10](#) uses the **Sqrt** method along with the **Pythagorean Theorem** to compute a new value for the Y component, which when combined with the new X component will produce a direction vector having the same **length** as before.

Adjust the signs of the X and Y components

Finally, the last two statements in [Listing 10](#) use the information gleaned earlier to cause the signs of the new X and Y components to match the signs of the original components.

A new direction vector with the same length in the same quadrant

By modifying the lengths of the X and Y components, the code in [Listing 10](#) causes the direction pointed to by the new vector to be different from the direction pointed to by the original direction vector.

By causing the X and Y components to have the same signs, the code in [Listing 10](#) causes the new direction vector to point into the same quadrant as before.

The Draw method

The **Sprite.Draw** method is shown in its entirety in [Listing 11](#).

Note:

Listing 11 . The Sprite.Draw method.

```
public void Draw(SpriteBatch spriteBatch) {  
    //Call the simplest available version of  
    // spriteBatch.Draw  
  
    spriteBatch.Draw(image, position, Color.White);  
} //end Draw method  
//-----  
-----//  
} //end Sprite class  
} //end namespace
```

This **Draw** method is essentially the same as the **Draw** method that I explained in an earlier module so it shouldn't require further explanation.

To avoid becoming confused, however, you should keep in mind that this program deals with the following three methods having the name **Draw** :

- Overridden Game.Draw method.
- Sprite.Draw method.
- SpriteBatch.Draw method.

The end of the Sprite class

[Listing 11](#) also signals the end of the **Sprite** class.

The Game1 class

The **Game1** class begins in [Listing 12](#).

Note:

Listing 12 . Beginning of the Game1 class.

```
namespace XNA0128Proj {  
  
    public class Game1 :  
Microsoft.Xna.Framework.Game {  
        GraphicsDeviceManager graphics;  
        SpriteBatch spriteBatch;  
  
        //Use the following values to set the size of  
the  
        // client area of the game window. The actual  
window  
        // with its frame is somewhat larger depending
```

```

on
    // the OS display options. On my machine with
its
    // current display options, these dimensions
    // produce a 1024x768 game window.
    int windowHeight = 1017;
    int windowHeight = 738;

    //This is the length of the greatest distance
in
    // pixels that any sprite will move in a
single
    // frame of the game loop.
    double maxVectorLength = 5.0;

    //References to the space rocks are stored in
this
    // List object.
    List<Sprite> rocks = new List<Sprite>();
    int numRocks = 24; //Number of rocks.
    //The following value should never exceed 60
moves
    // per second unless the default frame rate is
also
    // increased to more than 60 frames per
second.
    double maxRockSpeed = 50; //moves per second

    //References to the power pills are stored in
    // this List.
    List<Sprite> pills = new List<Sprite>();
    int numPills = 12; //Number of pills.
    double maxPillSpeed = 40; //moves per second

    //References to the UFOs are stored in this
List.
    List<Sprite> ufos = new List<Sprite>();

```

```

int numUfos = 6; //Max number of ufos
double maxUfoSpeed = 30;

//Random number generator. It is best to use a
single
// object of the Random class to avoid the
// possibility of using different streams that
// produce the same sequence of values.
//Note that the random.NextDouble() method
produces
// a pseudo-random value where the sequence of
values
// is uniformly distributed between 0.0 and
1.0.
Random random = new Random();

```

Instance variables

[Listing 12](#) declares several instance variables. Comments are provided to explain most of the instance variables. No explanation beyond the comments in [Listing 12](#) should be required.

The modified constructor

The constructor is shown in its entirety in [Listing 13](#).

Note:

Listing 13 . The modified constructor.

```

public Game1() { //constructor
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
}

```



```

        //Set the size of the game window.
        graphics.PreferredBackBufferWidth =
windowWidth;
        graphics.PreferredBackBufferHeight =
windowHeight;
    }//end constructor

```

Nothing in this constructor is new to this module. Therefore, no further explanation should be required.

The overridden LoadContent method

The overridden **LoadContent** method begins in [Listing 14](#).

Note:

Listing 14 . Beginning of the overridden LoadContent method.

```

        protected override void LoadContent() {
            spriteBatch = new
SpriteBatch(GraphicsDevice);

            for(int cnt = 0;cnt < numRocks;cnt++){
                rocks.Add(new
Sprite("Rock",Content,random));

                //Set the position of the current rock at
a
                // random location within the game window.
                rocks[cnt].Position = new Vector2(
                    (float)(windowWidth *
random.NextDouble()),

```

```

        (float)(windowHeight *
random.NextDouble()));

        //Get a direction vector for the current
rock.
        // Make both components positive to cause
the
        // vector to point down and to the right.
        rocks[cnt].Direction = DirectionVector(
            (float)maxVectorLength,
            (float)(maxVectorLength *
random.NextDouble()),
            false, //xNeg
            false); //yNeg

        //Notify the Sprite object of the size of
the
        // game window.
        rocks[cnt].WindowSize =
            new
Point(windowWidth, windowHeight);

        //Set the speed in moves per second for
the
        // current sprite to a random value
between
        // maxRockSpeed/2 and maxRockSpeed.
        rocks[cnt].Speed = maxRockSpeed/2
            + maxRockSpeed *
random.NextDouble()/2;
    } //end for loop

```

Instantiate all of the space rock sprites

The code in [Listing 14](#) uses a **for** loop to instantiate all of the space rocks and to prepare them to move from left to right, top to bottom across the game window.

Call the Sprite constructor

The **for** loop begins with a call to the **Sprite** constructor to construct a new object of the **Sprite** class.

Note: As mentioned earlier, you should pass a reference to the same **Random** object each time you make a call to the **Sprite** constructor.

Each new **Sprite** object's reference is added to the list referred to by the instance variable named **rocks** .

Set the property values

Once the new **Sprite** object is constructed, the object's reference is accessed and used to set the following property values:

- The **Position** property of each rock is set to a random position within the game window.
- The **Direction** property for each rock is set to a value obtained by calling the private **DirectionVector** method. (I will explain the **DirectionVector** method later.)
- The **WindowSize** property for each rock is set to the common **windowWidth** and **windowHeight** values.
- The **Speed** property for each rock is set to a bounded random value.

Except for the call to the **DirectionVector** method, the code in [Listing 14](#) is straightforward and should not require an explanation beyond the embedded comments. I will explain the **DirectionVector** method shortly.

Instantiate and set properties on the power pills and the UFOs

[Listing 15](#) uses essentially the same code to instantiate and set the property values on the power pill sprites and the UFO sprites.

Note:

Listing 15 . Instantiate and set properties on the power pills and the UFOs.

```
//Use the same process to instantiate all of
the
// power pills and cause them to move from
right
// to left, top to bottom.
for(int cnt = 0;cnt < numPills;cnt++) {
    pills.Add(new
Sprite("ball",Content,random));
    pills[cnt].Position = new Vector2(
        (float)(windowWidth *
random.NextDouble()),
        (float)(windowHeight *
random.NextDouble()));
    pills[cnt].Direction = DirectionVector(
        (float)maxVectorLength,
        (float)(maxVectorLength *
random.NextDouble()),
        true, //xNeg
        false); //yNeg
    pills[cnt].WindowSize =
        new
Point(windowWidth,windowHeight);
    pills[cnt].Speed = maxPillSpeed/2
        + maxPillSpeed *
random.NextDouble()/2;
} //end for loop

//Use the same process to instantiate all of
```

```

the
    // ufos and cause them to move from right to
left,
    // bottom to top.
    for(int cnt = 0;cnt < numUfos;cnt++) {
        ufos.Add(new
Sprite("ufo",Content,random));
        ufos[cnt].Position = new Vector2(
            (float)(windowWidth *
random.NextDouble()),
            (float)(windowHeight *
random.NextDouble()));
        ufos[cnt].Direction = DirectionVector(
            (float)maxVectorLength,
            (float)(maxVectorLength *
random.NextDouble()),
            true,//xNeg
            true);//yNeg
        ufos[cnt].WindowSize =
            new
Point(windowWidth,windowHeight);
        ufos[cnt].Speed = maxUfoSpeed/2
            + maxUfoSpeed *
random.NextDouble()/2;
    }//end for loop

    }//end LoadContent

```

The private **DirectionVector** method

The **DirectionVector** method shown in [Listing 16](#) is called each time the code in [Listing 14](#) and [Listing 15](#) needs to set the **Direction** property on a sprite. This method is declared private because it has no meaning outside the **Sprite** class.

Note:

Listing 16 . The private DirectionVector method.

```
private Vector2 DirectionVector(float vecLen,
                                float xLen,
                                Boolean negX,
                                Boolean negY){
    Vector2 result = new Vector2(xLen,0);
    result.Y = (float)Math.Sqrt(vecLen*vecLen
                                -
                                xLen*xLen);
    if(negX)
        result.X = -result.X;
    if(negY)
        result.Y = -result.Y;
    return result;
} //end DirectionVector
```

Return a direction vector

The **DirectionVector** method returns a direction vector as type **Vector2** given the length of the vector, the length of the X component of the vector, the sign of the X component, and the sign of the Y component.

The signs of the components

You should set **negX** and/or **negY** to true to cause them to be negative.

By adjusting the signs on the X and Y components, the vector can be caused to point into any of the four quadrants. The relationships between these two values and the direction of motion of the sprite are as follows:

- false, false = down and to the right.
- true, false = down and to the left

- true, true = up and to the left
- false, true = up and to the right

The code in [Listing 16](#) is very similar to the code that I explained earlier in [Listing 10](#), so the code in [Listing 16](#) should not require further explanation.

The overridden Update method

The overridden **Update** method is shown in its entirety in [Listing 17](#).

Note:

Listing 17 . Tell the sprites to move.

```
protected override void Update(GameTime
gameTime) {
    //Tell all the rocks in the list to move.
    for(int cnt = 0;cnt < rocks.Count;cnt++) {
        rocks[cnt].Move(gameTime);
    }//end for loop

    //Tell all the power pills in the list to
move.
    for(int cnt = 0;cnt < pills.Count;cnt++) {
        pills[cnt].Move(gameTime);
    }//end for loop

    //Tell all the ufos in the list to move.
    for(int cnt = 0;cnt < ufos.Count;cnt++) {
        ufos[cnt].Move(gameTime);
    }//end for loop

    base.Update(gameTime);
} //end Update method
```

Very simple code

Because the code to instantiate the **Sprite** objects and set their properties was placed in the **LoadContent** method in this program, the code in the overridden **Update** method is very simple.

The code in [Listing 17](#) uses **for** loops to access the references and call the **Move** method on every **Sprite** object.

To move or not to move, that is the question

As you learned earlier, when the **Move** method is called on an individual **Sprite** object, the sprite may or it may not actually move depending on the value of its **Speed** property and the elapsed time since its last actual move.

A characteristic of an object-oriented program

One of the characteristics of an object-oriented program is that the individual objects know how to behave with minimal supervision. In effect, a call to the **Sprite.Move** method in [Listing 17](#) tells the object to make its own decision and to move if it is time for it to move.

The overridden **Game.Draw** method

[Listing 18](#) shows the overridden **Game.Draw** method in its entirety.

Note:

Listing 18 . The overridden **Game.Draw** method.

```
protected override void Draw(GameTime
gameTime) {
    GraphicsDevice.Clear(Color.CornflowerBlue);
```



```

        spriteBatch.Begin();

        //Draw all rocks.
        for(int cnt = 0;cnt < rocks.Count;cnt++) {
            rocks[cnt].Draw(spriteBatch);
        }//end for loop

        //Draw all power pills.
        for(int cnt = 0;cnt < pills.Count;cnt++) {
            pills[cnt].Draw(spriteBatch);
        }//end for loop

        //Draw all ufos.
        for(int cnt = 0;cnt < ufos.Count;cnt++) {
            ufos[cnt].Draw(spriteBatch);
        }//end for loop

        spriteBatch.End();

        base.Draw(gameTime);
    }//end Draw method
    //-----
-----//
    }//end class
} //end namespace

```

Erase and redraw the entire game window

[Listing 18](#) begins by painting over everything in the game window with CornflowerBlue pixels. Then it uses **for** loops to access and call the **Sprite.Draw** method on every **Sprite** object.

Each call to a **Sprite** object's **Draw** method in [Listing 18](#) is a notification to the **Sprite** object that it should cause itself to be redrawn in the appropriate position with the appropriate image in the game window.

Note: This is another manifestation of an object knowing how to behave with minimal supervision. The overridden `Game.Draw` method doesn't know and doesn't care where the `Sprite` object should be positioned or what image it should draw to represent itself. The `Game.Draw` method simply knows that every `Sprite` object must redraw itself at the appropriate position with the appropriate image at this point in time. Decisions regarding position and image are left up to the `Sprite` object.

Regardless of whether or not a sprite has decided to move, it should cause itself to be redrawn in the game window because its image has just been replaced by a bunch of `CornflowerBlue` pixels.

Not the same approach as some other game engines

Some game engines take a different approach and in the interest of speed, redraw only those portions of the game window that have changed. This can lead to a great deal of complexity and it is often the responsibility of the game programmer to manage that complexity.

That is not the case with XNA. From the viewpoint of the XNA game programmer, the entire game window is redrawn once during every iteration of the game loop. This causes XNA to be easier to program than some other game engines. On the other hand, it might be argued that XNA sacrifices speed for simplicity.

The end of the program

[Listing 18](#) also signals the end of the overridden `Game.Draw` method, the end of the `Game1` class, and the end of the program.

Run the program

I encourage you to copy the code from [Listing 19](#) and [Listing 20](#). Use that code to create an XNA project. (You should be able to use any small image files as sprites.) Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Run my program

Click [here](#) to download a zip file containing my version of the program. Extract the folder named **XNA0128Proj** from the zip file and save it somewhere on your disk. Start **Visual C# 2010 Express** and select **Open Project...** from the **File** menu. Navigate to the project folder and select the file with the extension of **.sln**. This should cause the project to open and be ready to run or debug as described in the earlier module titled [Getting Started](#).

Summary

In this module, you learned how to add more sophistication to the **Sprite** class that was introduced in an earlier module.

What's next?

We're getting very close to being able to write a 2D arcade style game involving UFOs, space rocks, and power pills. There are just a few more tools that we need and I will show you how to create those tools in the upcoming modules.

The critical tools are:

- The ability to detect collisions between sprites.
- The ability to control the movement of one or more sprites using the keyboard or the mouse.

Once we have those tools, we can write a game where the challenge is to cause the UFO to successfully navigate across the game window without

colliding with a space rock, so I will concentrate on those two tools in the next few modules. (There are probably other games that we could also create using those tools.)

Beyond that, there are several other tools that will make it possible for us to create more sophisticated and interesting games:

- The ability to play sounds.
- The ability to create onscreen text.
- The ability to create a game with multiple levels, increasing difficulty at each level, scorekeeping, etc.

I may show you how to create some of those tools later in this series of modules. On the other hand, I may decide to leave that as an exercise for the student.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Xna0128-Improving the Sprite Class
- File: Xna0128.htm
- Published: 02/28/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation :: I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

Complete listings of the XNA program files discussed in this module are provided in [Listing 19](#) and [Listing 20](#).

Note:

Listing 19 . The class named Sprite for the project named XNA0128Proj.

```
/*Project XNA0128Proj
 * This file defines a Sprite class from which a
Sprite
 * object can be instantiated.

*****
*****/

using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;

namespace XNA0128Proj {
```

```

class Sprite {
    private Texture2D image;
    private Vector2 position = new Vector2(0,0);
    private Vector2 direction = new Vector2(0,0);
    private Point windowSize;
    private Random random;
    double elapsedTime;//in milliseconds
    //The following value is the inverse of speed
in
    // moves/msec expressed in msec/move.
    double elapsedTimeTarget;
    //-----
-----//

    //Position property accessor
    public Vector2 Position {
        get {
            return position;
        }
        set {
            position = value;
        }//end set
    }//end Position property accessor
    //-----
-----//

    //WindowSize property accessor
    public Point WindowSize {
        set {
            windowSize = value;
        }//end set
    }//end WindowSize property accessor
    //-----
-----//

    //Direction property accessor
    public Vector2 Direction {

```

```

        get {
            return direction;
        }
        set {
            direction = value;
        } //end set
    } //end Direction property accessor
    //-----
-----//

    //Speed property accessor. The set side should
be    // called with speed in moves/msec. The get
side    // returns speed moves/msec.
    public double Speed {
        get {
            //Convert from elapsed time in msec/move
to        // speed in moves/msec.
            return elapsedTimeTarget/1000;
        }
        set {
            //Convert from speed in moves/msec to
            // elapsed time in msec/move.
            elapsedTimeTarget = 1000/value;
        } //end set
    } //end Speed property accessor
    //-----
-----//

    //This constructor loads an image for the
sprite    // when it is instantiated. Therefore, it
requires    // an asset name for the image and a reference
to a

```

```

        // ContentManager object.
        //Requires a reference to a Random object.
Should
        // use the same Random object for all sprites
to
        // avoid getting the same sequence for
different
        // sprites.
        public Sprite(String assetName,
                        ContentManager contentManager,
                        Random random) {
            image = contentManager.Load<Texture2D>
(assetName);
            this.random = random;
        } //end constructor
        //-----
-----//

        //This method can be called to load a new
image
        // for the sprite.
        public void SetImage(String assetName,
                               ContentManager
contentManager) {
            image = contentManager.Load<Texture2D>
(assetName);
        } //end SetImage
        //-----
-----//

        //This method causes the sprite to move in the
        // direction of the direction vector if the
elapsed
        // time since the last move exceeds the
elapsed
        // time target based on the specified speed.
        public void Move(GameTime gameTime) {

```



```

        //Accumulate elapsed time since the last
move.
        elapsedTime +=
gameTime.ElapsedGameTime.Milliseconds;

        if(elapsedTime > elapsedTimeTarget){
            //It's time to make a move. Set the
elapsed
            // time to a value that will attempt to
produce
            // the specified speed on the average.
            elapsedTime -= elapsedTimeTarget;

            //Add the direction vector to the position
            // vector to get a new position vector.
            position =
Vector2.Add(position,direction);

            //Check for a collision with an edge of
the game
            // window. If the sprite reaches an edge,
cause
            // the sprite to wrap around and reappear
at the
            // other edge, moving at the same speed in
a
            // different direction within the same
quadrant
            // as before.
            if(position.X < -image.Width){
                position.X = windowSize.X;
                NewDirection();
            }//end if

            if(position.X > windowSize.X){
                position.X = -image.Width/2;

```

```

        NewDirection();
    }//end if

    if(position.Y < -image.Height) {
        position.Y = windowSize.Y;
        NewDirection();
    }//end if

    if(position.Y > windowSize.Y){
        position.Y = -image.Height / 2;
        NewDirection();
    }//end if on position.Y
} //end if on elapsed time
} //end Move
//-----
-----//

//This method determines the length of the
current
// direction vector along with the signs of
the X
// and Y components of the current direction
vector.
// It computes a new direction vector of the
same
// length with the X and Y components having
random
// lengths and the same signs.
//Note that random.NextDouble returns a
// pseudo-random value, uniformly distributed
// between 0.0 and 1.0.
private void NewDirection() {
    //Get information about the current
direction
    // vector.
    double length = Math.Sqrt(
                                direction.X *

```

```

direction.X +
                                direction.Y *
direction.Y);
    Boolean xNegative = (direction.X < 0)?
true:false;
    Boolean yNegative = (direction.Y < 0)?
true:false;

    //Compute a new X component as a random
portion of
    // the vector length.
    direction.X =
                                (float)(length *
random.NextDouble());
    //Compute a corresponding Y component that
will
    // keep the same vector length.
    direction.Y = (float)Math.Sqrt(length*length
-
direction.X*direction.X);

    //Set the signs on the X and Y components to
match
    // the signs from the original direction
vector.
    if(xNegative)
        direction.X = -direction.X;
    if(yNegative)
        direction.Y = -direction.Y;
} //end NewDirection
//-----
-----//

public void Draw(SpriteBatch spriteBatch) {
    //Call the simplest available version of
    // spriteBatch.Draw

```

```

spriteBatch.Draw(image, position, Color.White);
    }//end Draw method
    //-----
-----//
    }//end class
}//end namespace

```

Note:

Listing 20 . The class named Game1 for the project named XNA0128Proj.

```

/*Project XNA0128Proj
 * This project demonstrates how to integrate
space
 * rocks, power pills, and ufos in a program using
 * objects of a Sprite class. This could be the
 * beginnings of a space game.
 *
*****
***/
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using XNA0128Proj;

namespace XNA0128Proj {

    public class Game1 :
Microsoft.Xna.Framework.Game {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

```

```

    //Use the following values to set the size of
the
    // client area of the game window. The actual
window
    // with its frame is somewhat larger depending
on
    // the OS display options. On my machine with
its
    // current display options, these dimensions
    // produce a 1024x768 game window.
    int windowWidth = 1017;
    int windowHeight = 738;

    //This is the length of the greatest distance
in
    // pixels that any sprite will move in a
single
    // frame of the game loop.
    double maxVectorLength = 5.0;

    //References to the space rocks are stored in
this
    // List object.
    List<Sprite> rocks = new List<Sprite>();
    int numRocks = 24; //Number of rocks.
    //The following value should never exceed 60
moves
    // per second unless the default frame rate is
also
    // increased to more than 60 frames per
second.
    double maxRockSpeed = 50; //frames per second

    //References to the power pills are stored in
    // this List.
    List<Sprite> pills = new List<Sprite>();
    int numPills = 12; //Number of pills.

```

```

    double maxPillSpeed = 40;//moves per second

    //References to the UFOs are stored in this
List.
    List<Sprite> ufos = new List<Sprite>();
    int numUfos = 6;//Max number of ufos
    double maxUfoSpeed = 30;

    //Random number generator. It is best to use a
single
    // object of the Random class to avoid the
    // possibility of using different streams that
    // produce the same sequence of values.
    //Note that the random.NextDouble() method
produces
    // a pseudo-random value where the sequence of
values
    // is uniformly distributed between 0.0 and
1.0.
    Random random = new Random();
    //-----
-----//

    public Game1() {//constructor
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        //Set the size of the game window.
        graphics.PreferredBackBufferWidth =
windowWidth;
        graphics.PreferredBackBufferHeight =
windowHeight;
    }//end constructor
    //-----
-----//

    protected override void Initialize() {

```

```

        //No initialization required.
        base.Initialize();
    }//end Initialize
    //-----
-----//

    protected override void LoadContent() {
        spriteBatch = new
SpriteBatch(GraphicsDevice);

        //Instantiate all of the rocks and cause
them to
        // move from left to right, top to
        // bottom. Pass a reference to the same
Random
        // object to all of the sprites.
        for(int cnt = 0;cnt < numRocks;cnt++){
            rocks.Add(new
Sprite("Rock",Content,random));

            //Set the position of the current rock at
a
            // random location within the game window.
            rocks[cnt].Position = new Vector2(
                (float)(windowWidth *
random.NextDouble()),
                (float)(windowHeight *
random.NextDouble()));

            //Get a direction vector for the current
rock.
            // Make both components positive to cause
the
            // vector to point down and to the right.
            rocks[cnt].Direction = DirectionVector(
                (float)maxVectorLength,
                (float)(maxVectorLength *

```

```

random.NextDouble()),
        false, //xNeg
        false); //yNeg

        //Notify the Sprite object of the size of
the
        // game window.
        rocks[cnt].WindowSize =
            new
Point(windowWidth, windowHeight);

        //Set the speed in moves per second for
the
        // current sprite to a random value
between
        // maxRockSpeed/2 and maxRockSpeed.
        rocks[cnt].Speed = maxRockSpeed/2
            + maxRockSpeed *
random.NextDouble()/2;
    } //end for loop

    //Use the same process to instantiate all of
the
    // power pills and cause them to move from
right
    // to left, top to bottom.
    for(int cnt = 0; cnt < numPills; cnt++) {
        pills.Add(new
Sprite("ball", Content, random));
        pills[cnt].Position = new Vector2(
            (float)(windowWidth *
random.NextDouble()),
            (float)(windowHeight *
random.NextDouble()));
        pills[cnt].Direction = DirectionVector(
            (float)maxVectorLength,
            (float)(maxVectorLength *

```



```

random.NextDouble()),
        true, //xNeg
        false); //yNeg
        pills[cnt].WindowSize =
            new
Point(windowWidth, windowHeight);
        pills[cnt].Speed = maxPillSpeed/2
            + maxPillSpeed *
random.NextDouble()/2;
    } //end for loop

    //Use the same process to instantiate all of
the
    // ufos and cause them to move from right to
left,
    // bottom to top.
    for(int cnt = 0; cnt < numUfos; cnt++) {
        ufos.Add(new
Sprite("ufo", Content, random));
        ufos[cnt].Position = new Vector2(
            (float)(windowWidth *
random.NextDouble()),
            (float)(windowHeight *
random.NextDouble()));
        ufos[cnt].Direction = DirectionVector(
            (float)maxVectorLength,
            (float)(maxVectorLength *
random.NextDouble()),
            true, //xNeg
            true); //yNeg
        ufos[cnt].WindowSize =
            new
Point(windowWidth, windowHeight);
        ufos[cnt].Speed = maxUfoSpeed/2
            + maxUfoSpeed *
random.NextDouble()/2;
    } //end for loop

```

```

        }//end LoadContent
        //-----
-----//

        //This method returns a direction vector given
the
        // length of the vector, the length of the
        // X component, the sign of the X component,
and the
        // sign of the Y component. Set negX and/or
negY to
        // true to cause them to be negative. By
adjusting
        // the signs on the X and Y components, the
vector
        // can be caused to point into any of the four
        // quadrants.
        private Vector2 DirectionVector(float vecLen,
                                         float xLen,
                                         Boolean negX,
                                         Boolean negY){
            Vector2 result = new Vector2(xLen,0);
            result.Y = (float)Math.Sqrt(vecLen*vecLen
                                         -
xLen*xLen);
            if(negX)
                result.X = -result.X;
            if(negY)
                result.Y = -result.Y;
            return result;
        }//end DirectionVector
        //-----
-----//

        protected override void UnloadContent() {
            //No content unload required.

```

```

        }//end unloadContent
        //-----
-----//

        protected override void Update(GameTime
gameTime) {
            //Tell all the rocks in the list to move.
            for(int cnt = 0;cnt < rocks.Count;cnt++) {
                rocks[cnt].Move(gameTime);
            }//end for loop

            //Tell all the power pills in the list to
move.
            for(int cnt = 0;cnt < pills.Count;cnt++) {
                pills[cnt].Move(gameTime);
            }//end for loop

            //Tell all the ufos in the list to move.
            for(int cnt = 0;cnt < ufos.Count;cnt++) {
                ufos[cnt].Move(gameTime);
            }//end for loop

            base.Update(gameTime);
        }//end Update method
        //-----
-----//

        protected override void Draw(GameTime
gameTime) {
            GraphicsDevice.Clear(Color.CornflowerBlue);

            spriteBatch.Begin();

            //Draw all rocks.
            for(int cnt = 0;cnt < rocks.Count;cnt++) {
                rocks[cnt].Draw(spriteBatch);
            }//end for loop

```

```

        //Draw all power pills.
        for(int cnt = 0;cnt < pills.Count;cnt++) {
            pills[cnt].Draw(spriteBatch);
        }//end for loop

        //Draw all ufos.
        for(int cnt = 0;cnt < ufos.Count;cnt++) {
            ufos[cnt].Draw(spriteBatch);
        }//end for loop

        spriteBatch.End();

        base.Draw(gameTime);
    }//end Draw method
    //-----
-----//
    }//end class
} //end namespace

```

-end-

Xna0130-Collision Detection

Learn how to design and create a Sprite class that provides collision detection. Also learn how to write an XNA program that takes advantage of that collision detection capability.

Revised: Mon May 09 18:37:13 CDT 2016

This page is part of a Book titled [XNA Game Studio](#).

Table of Contents

- [Table of Contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [Another improvement](#)
 - [Definition of a collision](#)
- [Preview](#)
 - [An epic battle](#)
 - [Spider and ladybug motion](#)
 - [The spiders don't die easily](#)
 - [The death blow](#)
 - [Output screen images](#)
 - [The top image](#)
 - [The middle image](#)
 - [The bottom image](#)

- [Discussion and sample code](#)
 - [The Sprite class](#)
 - [A new Image property accessor method](#)
 - [A modified constructor](#)
 - [A modified SetImage method](#)
 - [A new method named GetRectangle](#)
 - [A new method named IsCollision](#)
 - [What is a collision?](#)
 - [What is the overall behavior?](#)
 - [The IsCollision method code](#)
 - [Test for a collision with other sprites](#)
 - [A while loop](#)
 - [The first return statement](#)
 - [The second return statement](#)
 - [The end of the Sprite class](#)
 - [The Game1 class](#)
 - [The overridden LoadContent method](#)
 - [A background sprite with a spider web image](#)
 - [Instantiate the spider and ladybug Sprite objects](#)
 - [The overridden Update method](#)
 - [A for loop that controls the collision testing](#)
 - [Not every spider is tested during each iteration](#)
 - [Test for a collision](#)
 - [Return a reference to a spider or null](#)
 - [If a collision was detected...](#)
 - [Why change the spider's position?](#)
 - [Spiders don't die easily](#)
 - [The Name property](#)

- [Killing the brown spider](#)
- [Is a Dispose Method needed?](#)
- [The end of the Update method](#)
- [The overridden Game.Draw method](#)
 - [The end of the program](#)
- [Run the program](#)
- [Run my_program](#)
- [Summary](#)
- [What's next?](#)
- [Miscellaneous](#)
- [Complete program listing](#)

Preface

This module is one in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

An earlier module titled [Getting Started](#) provided information on how to get started programming with Microsoft's XNA Game Studio.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Screen output at three different times while the program was running.

Listings

- [Listing 1](#). A new Image property accessor method in the Sprite class.
- [Listing 2](#). A modified constructor.
- [Listing 3](#). A modified SetImage method.
- [Listing 4](#). A new method named GetRectangle.
- [Listing 5](#). Beginning of a new IsCollision method.
- [Listing 6](#). Test for a collision with other sprites.
- [Listing 7](#). Beginning of the overridden LoadContent method of the Game1 class.
- [Listing 8](#). Instantiate the spider and ladybug Sprite objects.
- [Listing 9](#). Beginning of the overridden Update method.
- [Listing 10](#). Beginning of a for loop that controls the collision testing.
- [Listing 11](#). Test for a collision.
- [Listing 12](#). Spiders don't die easily.
- [Listing 13](#). The overridden Game.Draw method.
- [Listing 14](#). The Sprite class for the project named XNA0130Proj.
- [Listing 15](#). The Game1 class for the project named XNA0130Proj.

General background information

While studying the past couple of modules, you have learned how to create and then to improve a class named **Sprite** from which you can instantiate objects that behave as sprites.

Another improvement

In this module, we will once again improve the **Sprite** class, this time adding the capability for a sprite to determine if it has collided with another sprite. This capability is critical for game development in many areas.

Definition of a collision

Our definition of a collision is based on the bounding rectangles for the images that represent the two sprites. If the rectangles intersect in their current positions, a collision is deemed to have occurred. If they don't intersect, there is no collision.

Preview

This program is not yet a game because it doesn't provide player interaction. Instead it is a demonstration program that demonstrates sprite collision detection in a rather interesting way.

[Figure 1](#) shows a screen snapshot at three different points in time while the program was running.

An epic battle

The demonstration program chronicles a battle between spiders and ladybugs. When the program starts, there are 200 black widow spiders and five ladybugs on the web in the game window. (See the top image in [Figure 1](#).)

Spider and ladybug motion

The spiders move at different speeds in different directions but generally toward the southeast. When a spider goes outside the game window on the right side or the bottom, it reappears on the left side or the top.

The ladybugs also move at different speeds in different directions but generally toward the northwest. When a ladybug goes outside the game window on the left side or the top, it reappears on the right side or on the bottom.

The spiders don't die easily

When a ladybug collides with a black widow spider, the spider disappears but is reincarnated as a green spider 128 pixels to the right of its original position.

When a ladybug collides with a green spider, the spider disappears but is reincarnated again as a small brown spider 128 pixels to the right of its original position.

The death blow

Finally, when a ladybug collides with a brown spider, the spider is eaten and is removed from the population of spiders. Therefore, all of the spiders eventually disappear and the ladybugs continue marching on victorious.

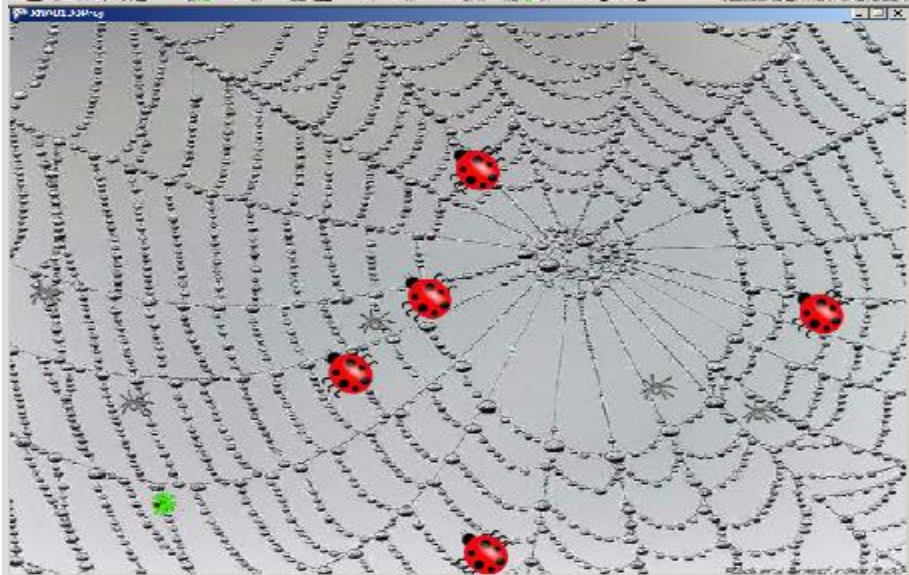
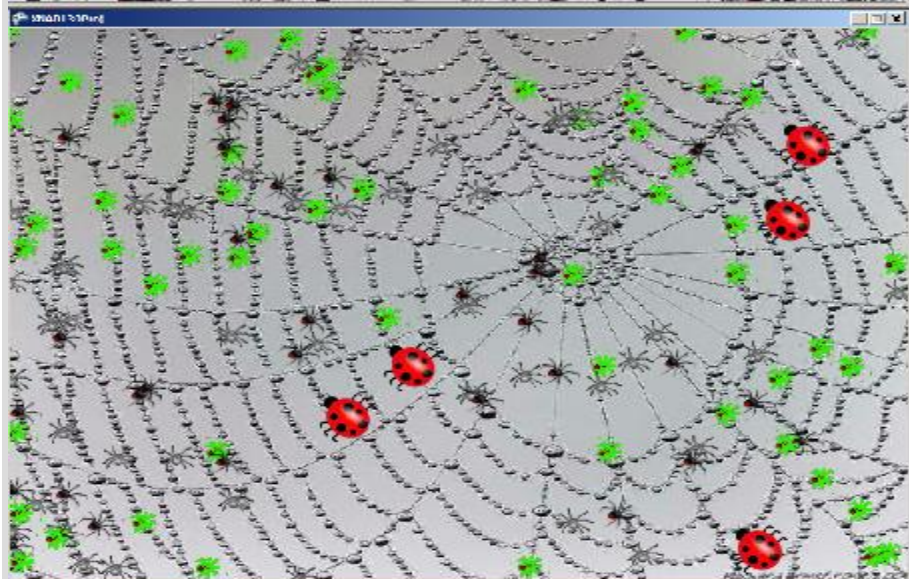
Output screen images

The top image

The top image in [Figure 1](#) was taken shortly after the program began running. Therefore, the game window was mostly populated with ladybugs and black widow spiders. There are a few green spiders and I believe I see one brown spider at the very bottom border near the right side.

Note:

Figure 1 . Screen output at three different times while the program was running.



The middle image

The middle image in [Figure 1](#) shows the program output after the program has been running for awhile. At this point in time, the game window is mainly populated with ladybugs, green spiders, and brown spiders. However, there are a few black widow spiders still in the picture.

The bottom image

The bottom image shows the program output at an even later point in time. At this point, the game window is populated with ladybugs, one green spider and a few brown spiders. Given enough time and a little luck, the ladybugs will collide with and destroy even these remaining spiders.

Discussion and sample code

As usual, I will discuss the program in fragments, beginning with the **Sprite** class. Furthermore, I will discuss only those portions of the **Sprite** class that are different from the versions of the **Sprite** class that I explained in earlier modules.

A complete listing of the **Sprite** class is provided in [Listing 14](#) and a complete listing of the **Game1** class is provided in [Listing 15](#) near the end of the module.

The Sprite class

A new Image property accessor method

Scanning down the **Sprite** class definition in [Listing 14](#), the first thing that we find that is new to this version is a new read-only **Image** property accessor method. The method is shown in [Listing 1](#).

Note:

Listing 1 . A new Image property accessor method in the Sprite class.

```
//Image property accessor - new to this
version.
public Texture2D Image {
    get {
        return image;
    } //end get
} //end Image property accessor
```

There is nothing unusual about this property accessor method, so it shouldn't require further explanation. You will learn why it is needed later when I explain the **Game1** class.

A modified constructor

The second statement from the end in [Listing 2](#) was added to the constructor from the previous version of the **Sprite** class.

Note:

Listing 2 . A modified constructor.

```
public Sprite(String assetName,
               ContentManager contentManager,
               Random random) {
```

```
        image = contentManager.Load<Texture2D>
(assetName);
        image.Name = assetName;
        this.random = random;
    }//end constructor
```

The new statement assigns the **assetName** to the **Name** property of the **Texture2D** object that represents the sprite's image. You will see why this modification was needed later.

A modified SetImage method

As with the modified constructor, the last statement in [Listing 3](#) was added to the **SetImage** method from the previous version of the **Sprite** class.

Note:

Listing 3 . A modified SetImage method.

```
        public void SetImage(String assetName,
                                ContentManager
contentManager) {
            image = contentManager.Load<Texture2D>
(assetName);
            image.Name = assetName;
        }//end SetImage
```

Once again, the new statement assigns the **assetName** to the **Name** property of the **Texture2D** object that represents the sprite's image. You will also see why this modification was needed later.

A new method named **GetRectangle**

[Listing 4](#) shows a new method named **GetRectangle** .

Note:

Listing 4 . A new method named **GetRectangle**.

```
public Rectangle GetRectangle() {  
    return new Rectangle((int)(position.X),  
                        (int)(position.Y),  
                        image.Width,  
                        image.Height);  
} //end GetRectangle
```

This method returns the current rectangle occupied by the sprite's image as type **Rectangle** . This rectangle is needed for the collision detection process. The code in [Listing 4](#) is straightforward and shouldn't require further explanation.

A new method named **IsCollision**

This version of the **Sprite** class defines a new method named **IsCollision** . The purpose of this new method is to detect a collision between this sprite and some other sprite.

What is a collision?

A collision is called if the rectangle containing this **Sprite** object's image intersects the rectangle containing a target sprite's image.

What is the overall behavior?

This method receives a list of **Sprite** objects as an incoming parameter. It tests for a collision with each sprite in the list beginning with the sprite at the head of the list. If it detects a collision with a sprite, it stops testing immediately and returns a reference to the **Sprite** object for which it found the collision. If it doesn't find a collision with any sprite in the list, it returns null.

The `IsCollision` method code

The new method named **IsCollision** begins in [Listing 5](#).

Note:

Listing 5 . Beginning of a new `IsCollision` method.

```
public Sprite IsCollision(List<Sprite> target)
{
    Rectangle thisRectangle =
        new Rectangle((int)
(position.X),
                    (int)
(position.Y),
image.Width,
image.Height);
```

The code in [Listing 5](#) constructs a new **Rectangle** object that describes the rectangular area currently occupied by the sprite's image relative to the upper left corner of the game window. The **Rectangle** object's reference is saved in the local variable named **thisRectangle** .

Test for a collision with other sprites

[Listing 6](#) begins by declaring a pair of local variables named **targetRectangle** and **cnt** .

The variable named **targetRectangle** will be used to store a reference to a rectangular area that currently contains a target sprite. The variable named **cnt** will be used as a loop counter.

Note:

Listing 6 . Test for a collision with other sprites.

```
Rectangle targetRectangle;
int cnt = 0;

while(cnt < target.Count){
    targetRectangle =
target[cnt].GetRectangle();
if(thisRectangle.Intersects(targetRectangle)){
    return target[cnt];
} //end if
cnt++;
} //end while loop

return null; //no collision detected
} //end IsCollision
```

A while loop

Then [Listing 6](#) executes a **while** loop that:

- Calls the **GetRectangle** method on the next **Sprite** object in the list (see [Listing 4](#)).

- Tests to determine if this sprite's rectangle intersects the target sprite's rectangle.
- Returns a reference to the target **Sprite** object if a collision is detected; keeps looping otherwise.

The first return statement

Executing a **return** statement in the middle of the **while** loop terminates the loop and also terminates the method.

The second return statement

If no collision is detected, the loop terminates when all of the **Sprite** objects in the list have been tested.

At that point, [Listing 6](#) executes a different **return** statement to return **null** and terminate the method. The **null** value is returned in place of a reference to a **Sprite** object to signal that no collision was detected.

The end of the Sprite class

[Listing 6](#) signals the end of the **IsCollision** method, which is also the end of the changes made to the **Sprite** class. Once again, you will find a complete listing of the new **Sprite** class in [Listing 14](#) near the end of the module.

The Game1 class

I will explain only those portions of the **Game1** class that are substantially new to this program. You will find a complete listing of the **Game1** class in [Listing 15](#) near the end of the module.

The overridden LoadContent method

The only code that is substantially new prior to the **LoadContent** method is the declaration of an instance variable of type **Sprite** named **spiderWeb** . Therefore, I will skip down and begin my discussion with the overridden **LoadContent** method, which begins in [Listing 7](#).

Note:

Listing 7 . Beginning of the overridden LoadContent method of the Game1 class.

```
protected override void LoadContent() {  
    spriteBatch = new  
    SpriteBatch(GraphicsDevice);  
  
    //Create a sprite for the background image.  
    spiderWeb =  
        new  
    Sprite("spiderwebB",Content,random);  
    spiderWeb.Position = new Vector2(0f,0f);  
}
```

A background sprite with a spider web image

The last two statements in [Listing 7](#) instantiate a new **Sprite** object and load it with an image of the spider web shown in [Figure 1](#).

The purpose of this sprite is to serve as a background image. Before adding the image of the spider web to the **Content** folder during the design phase, I used an external program to scale the image to the same size as the game window established by the constructor in [Listing 15](#).

The code in [Listing 7](#) positions the upper left corner of the sprite at the upper left corner of the game window so that it just fills the game window as shown in [Figure 1](#).

Instantiate the spider and ladybug Sprite objects

The code in [Listing 8](#) instantiates all of the spider and ladybug **Sprite** objects and sets their properties.

Note:

Listing 8 . Instantiate the spider and ladybug Sprite objects.

```
//Instantiate all of the spiders and cause
them to
// move from left to right, top to
// bottom. Pass a reference to the same
Random
// object to all of the sprites.
for(int cnt = 0;cnt < numSpiders;cnt++) {
    spiders.Add(
        new
Sprite("blackWidowSpider",Content,random));

        //Set the position of the current spider
at a
        // random location within the game window.
        spiders[cnt].Position = new Vector2(
            (float)(windowWidth *
random.NextDouble()),
            (float)(windowHeight *
random.NextDouble()));

        //Get a direction vector for the current
spider.
        // Make both components positive to cause
the
        // vector to point down and to the right.
        spiders[cnt].Direction = DirectionVector(
            (float)maxVectorLength,
            (float)(maxVectorLength *
            (float)random.NextDouble()));
```

```

random.NextDouble()),
        false, //xNeg
        false); //yNeg

        //Notify the spider object of the size of
the
        // game window.
        spiders[cnt].WindowSize =
            new
Point(windowWidth, windowHeight);

        //Set the speed in moves per second for
the
        // current spider to a random value
between
        // maxSpiderSpeed/2 and maxSpiderSpeed.
        spiders[cnt].Speed = maxSpiderSpeed / 2
            + maxSpiderSpeed *
random.NextDouble() / 2;
    } //end for loop

    //Use the same process to instantiate all of
the
    // ladybugs and cause them to move from
right to
    // left, bottom to top.
    for(int cnt = 0; cnt < numLadybugs; cnt++) {
        ladybugs.Add(
            new
Sprite("ladybug", Content, random));
        ladybugs[cnt].Position = new Vector2(
            (float)(windowWidth *
random.NextDouble()),
            (float)(windowHeight *
random.NextDouble()));
        ladybugs[cnt].Direction = DirectionVector(
            (float)maxVectorLength,

```

```

        (float)(maxVectorLength *
random.NextDouble()),
        true,//xNeg
        true);//yNeg
        ladybugs[cnt].WindowSize =
            new
Point(windowWidth,windowHeight);
        ladybugs[cnt].Speed = maxLadybugSpeed / 2
            + maxLadybugSpeed *
random.NextDouble() / 2;
    }//end for loop

} //end LoadContent

```

The code in [Listing 8](#) is essentially the same as code that I explained in an earlier module so no explanation beyond the embedded comments should be necessary.

The overridden Update method

The overridden **Update** method begins in [Listing 9](#).

Note:

Listing 9 . Beginning of the overridden Update method.

```

protected override void Update(GameTime
gameTime) {
    //Tell all the spiders in the list to move.
    for(int cnt = 0;cnt < spiders.Count;cnt++) {
        spiders[cnt].Move(gameTime);
    } //end for loop
}

```

```
//Tell all the ladybugs in the list to move.  
for(int cnt = 0;cnt < ladybugs.Count;cnt++)  
{  
    ladybugs[cnt].Move(gameTime);  
}//end for loop
```

The code in [Listing 9](#) is essentially the same as code that I have explained in earlier modules, so no further explanation should be necessary.

A for loop that controls the collision testing

[Listing 10](#) shows the beginning of a **for** loop that causes each ladybug **Sprite** object to test for a collision with the spiders in the list of spiders once during each iteration of the game loop.

If a ladybug detects a collision with a spider, the action described [earlier](#) is taken.

Note:

Listing 10 . Beginning of a for loop that controls the collision testing.

```
for(int cnt = 0;cnt < ladybugs.Count;cnt++)  
{
```

Not every spider is tested during each iteration

If a ladybug detects a collision with a spider, the remaining spiders in the list are not tested by that ladybug during that iteration of the game loop. In other words, if a ladybug's rectangle intersects the rectangles belonging to

two or more spiders, only the spider closest to the top of the list will register a collision.

Test for a collision

The code that tests for a collision is shown in [Listing 11](#).

Note:

Listing 11 . Test for a collision.

```
        //Test for a collision between this
ladybug and
        // all of the spiders in the list of
spiders.
        Sprite target =
ladybugs[cnt].IsCollision(spiders);

        if(target != null) {
            //There was a collision. Cause the
spider to
            // move 128 pixels to the right.
            target.Position =
                new Vector2(target.Position.X
+ 128,
target.Position.Y);
```

The first statement in [Listing 11](#) calls the **IsCollision** method on the current ladybug object, passing a reference to the list of spiders as a parameter. As you learned earlier, this will cause the ladybug object to test each spider in the list for a collision until either a collision is found or the list is exhausted.

Return a reference to a spider or null

When the **IsCollision** method returns and control moves to the beginning of the **if** statement, the variable named **target** will either contain **null** , (meaning that no collision was detected), or will contain a reference to the spider object involved in a collision.

If a collision was detected...

The body of the **if** statement shown in [Listing 11](#) is executed if a collision was detected and the **target** variable does not contain null. In this case, the X component of the spider's position vector is increased by a value of 128. This will cause the spider to move 128 pixels to the right the next time it is drawn.

Why change the spider's position?

This change in position is necessary to prevent the same ladybug from registering a collision with the same spider during the next iteration of the game loop. The forward movement of the spiders and the ladybugs each time they move is less than the dimensions of the intersecting rectangles. Therefore, without a purposeful shift in position of either the ladybug or the spider, the pair would continue to register collisions until the rectangles finally separate from one another.

Spiders don't die easily

As you read in the [Preview](#) section, if a black widow spider collides with a ladybug, it simply moves 128 pixels to the right and changes into a green spider. If a green spider collides with a ladybug, it simply moves 128 pixels to the right and changes into a brown spider. Finally, if a brown spider collides with a ladybug, it is eaten and removed from the population of spiders. This is accomplished by the code in [Listing 12](#).

Note:

Listing 12 . Spiders don't die easily.

```
        //If the collision was with a black
widow
        // spider, cause it to reincarnate into
a
        // green spider.
        if(target.Image.Name ==
"blackWidowSpider") {
target.SetImage("greenspider",Content);
        }
        //If the collision was with a green
spider,
        // cause it to reincarnate into a brown
        // spider.
        else if(target.Image.Name ==
"greenspider") {
target.SetImage("brownSpider",Content);
        }
        //If the collision was with a brown
spider,
        // it gets eaten. Remove it from the
list of
        // spiders.
        else if(target.Image.Name ==
"brownSpider") {
            spiders.Remove(target);
        }// end else-if
    }//end if
} //end for loop

    base.Update(gameTime);
} //end Update method
```

The code in [Listing 12](#) is relatively straightforward.

The Name property

The first statement in [Listing 12](#) shows why I needed to set an identifiable string value into the **Name** property of the **Texture2D** object referred to by **image** in [Listing 2](#) and [Listing 3](#). I needed a way to determine the type of spider involved in each collision so that I could take the appropriate action when a collision was detected.

Killing the brown spider

The call to the **spiders.Remove** method in [Listing 12](#) removes the target **Sprite** object's reference from the list of spiders. Since this is the only reference to the **Sprite** object, this action should make the memory occupied by the **Sprite** object eligible for garbage collection.

Is a Dispose Method needed?

However, I'm not certain that garbage collection is sufficient to free up all of the resources owned by the now defunct object and its image. It might also be advisable to use a [Dispose Method](#). At the time I am writing this, I simply don't know.

The end of the Update method

[Listing 12](#) also signals the end of the overridden **Update** method.

The overridden Game.Draw method

The overridden **Game.Draw** method is shown in its entirety in [Listing 13](#).

Note:

Listing 13 . The overridden Game.Draw method.

```
protected override void Draw(GameTime
gameTime) {

    spriteBatch.Begin();

    spiderWeb.Draw(spriteBatch); //draw
background

    //Draw all spiders.
    for(int cnt = 0; cnt < spiders.Count; cnt++) {
        spiders[cnt].Draw(spriteBatch);
    } //end for loop

    //Draw all ladybugs.
    for(int cnt = 0; cnt < ladybugs.Count; cnt++)
{
    ladybugs[cnt].Draw(spriteBatch);
} //end for loop

    spriteBatch.End();

    base.Draw(gameTime);
} //end Draw method
//-----
-----//
} //end class
} //end namespace
```

There is nothing new in [Listing 13](#). However, the call to **spiderWeb.Draw** is a little different from the code that you have seen in the overridden **Game.Draw** methods in the last couple of modules. This code calls the

Sprite.Draw method, which in turn calls the **SpriteBatch.Draw** method to draw the image of the spider web as a background image in [Figure 1](#).

The end of the program

[Listing 13](#) also signals the end of the overridden **Game.Draw** method, the end of the **Game1** class, and the end of the program.

Run the program

I encourage you to copy the code from [Listing 14](#) and [Listing 15](#). Use that code to create an XNA project. You should be able to find and download suitable image files from the web. Any small images can be used as substitutes for the spiders and the ladybugs. A larger image can be used for the background.

Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make sure that you can explain why your changes behave as they do.

Run my program

Click [here](#) to download a zip file containing my version of the program. Extract the folder named **XNA0130Proj** from the zip file and save it somewhere on your disk. Start **Visual C# 2010 Express** and select **Open Project...** from the **File** menu. Navigate to the project folder and select the file with the extension of **.sln**. This should cause the project to open and be ready to run or debug as described in the earlier module titled [Getting Started](#).

Summary

You learned how to design and create a **Sprite** class that provides collision detection. You also learned how to write an XNA program that takes advantage of that collision detection capability.

What's next?

The one critical element that is preventing us from using the **Sprite** class to create a 2D arcade style game is the ability of the user to control the motion of one or more sprites using keyboard and/or mouse input. That will be the topic for the next module.

Once we have that tool, we can write a game where the challenge is to prevent the spiders from successfully navigating across the game window without being eaten by a ladybug. Or, we could write a game where the challenge is to cause the ladybugs to successfully navigate across the game window without being bitten by a spider.

Beyond that, there are several other tools that will make it possible for us to create more sophisticated and interesting games:

- The ability to play sounds.
- The ability to create onscreen text.
- The ability to create a game with multiple levels and increasing difficulty at each level, scorekeeping, etc.

I will show you how to create some of those tools later in this series of modules and will leave others as an exercise for the student.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Xna0130-Collision Detection
- File: Xna0130.htm
- Published: 02/28/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

Complete listings of the program files discussed in this module are provided in [Listing 14](#) and [Listing 15](#) below.

Note:

Listing 14 . The Sprite class for the project named XNA0130Proj.

```
/*Project XNA0130Proj
 * This file defines a Sprite class from which a
Sprite
 * object can be instantiated. This version
supports
 * collision detection based on intersecting
rectangles.
```

```

*****
*****/

using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;

namespace XNA0130Proj {
    class Sprite {
        private Texture2D image;
        private Vector2 position = new Vector2(0,0);
        private Vector2 direction = new Vector2(0,0);
        private Point windowSize;
        private Random random;
        double elapsedTime;//in milliseconds
        //The following value is the inverse of speed
in
        // moves/msec expressed in msec/move.
        double elapsedTimeTarget;
        //-----
        -----//

        //Image property accessor - new to this
version.
        public Texture2D Image {
            get {
                return image;
            }//end get
        }//end Image property accessor
        //-----
        -----//

        //Position property accessor
        public Vector2 Position {
            get {

```



```

        return position;
    }
    set {
        position = value;
    } //end set
} //end Position property accessor
//-----
-----//

//WindowSize property accessor
public Point WindowSize {
    set {
        windowSize = value;
    } //end set
} //end WindowSize property accessor
//-----
-----//

//Direction property accessor
public Vector2 Direction {
    get {
        return direction;
    }
    set {
        direction = value;
    } //end set
} //end Direction property accessor

//-----
-----//

//Speed property accessor. The set side should
be
// called with speed in moves/msec. The get
side
// returns speed moves/msec.
public double Speed {

```

```

        get {
            //Convert from elapsed time in msec/move
to
            // speed in moves/msec.
            return elapsedTimeTarget/1000;
        }
        set {
            //Convert from speed in moves/msec to
            // elapsed time in msec/move.
            elapsedTimeTarget = 1000/value;
        } //end set
    } //end Speed property accessor
    //-----
    -----//

    //This constructor loads an image for the
sprite
    // when it is instantiated. Therefore, it
requires
    // an asset name for the image and a reference
to a
    // ContentManager object.
    //Requires a reference to a Random object.
Should
    // use the same Random object for all sprites
to
    // avoid getting the same sequence for
different
    // sprites.
    public Sprite(String assetName,
                  ContentManager contentManager,
                  Random random) {
        image = contentManager.Load<Texture2D>
(assetName);

        image.Name = assetName; //new to this version
        this.random = random;
    } //end constructor

```

```

//-----
-----//

//This method can be called to load a new
image
// for the sprite.
public void SetImage(String assetName,
                    ContentManager
contentManager) {
    image = contentManager.Load<Texture2D>
(assetName);
    image.Name = assetName;//new to this version
} //end SetImage
//-----
-----//

//This method causes the sprite to move in the
// direction of the direction vector if the
elapsed
// time since the last move exceeds the
elapsed
// time target based on the specified speed.
public void Move(GameTime gameTime) {
    //Accumulate elapsed time since the last
move.
    elapsedTime +=
gameTime.ElapsedGameTime.Milliseconds;

    if(elapsedTime > elapsedTimeTarget){
        //It's time to make a move. Set the
elapsed
        // time to a value that will attempt to
produce
        // the specified speed on the average.
        elapsedTime -= elapsedTimeTarget;
    }
}

```

```

        //Add the direction vector to the position
        // vector to get a new position vector.
        position =
Vector2.Add(position,direction);

        //Check for a collision with an edge of
the game
        // window. If the sprite reaches an edge,
cause
        // the sprite to wrap around and reappear
at the
        // other edge, moving at the same speed in
a
        // different direction within the same
quadrant
        // as before.
        if(position.X < -image.Width){
            position.X = windowSize.X;
            NewDirection();
        }//end if

        if(position.X > windowSize.X){
            position.X = -image.Width/2;
            NewDirection();
        }//end if

        if(position.Y < -image.Height) {
            position.Y = windowSize.Y;
            NewDirection();
        }//end if

        if(position.Y > windowSize.Y){
            position.Y = -image.Height / 2;
            NewDirection();
        }//end if on position.Y
    }//end if on elapsed time
} //end Move

```

```

//-----
-----//

//This method determines the length of the
current
// direction vector along with the signs of
the X
// and Y components of the current direction
vector.
// It computes a new direction vector of the
same
// length with the X and Y components having
random
// lengths and the same signs.
//Note that random.NextDouble returns a
// pseudo-random value, uniformly distributed
// between 0.0 and 1.0.
private void NewDirection() {
    //Get information about the current
direction
    // vector.
    double length = Math.Sqrt(
                                direction.X *
direction.X +
                                direction.Y *
direction.Y);
    Boolean xNegative = (direction.X < 0)?
true:false;
    Boolean yNegative = (direction.Y < 0)?
true:false;

    //Compute a new X component as a random
portion of
    // the vector length.
    direction.X =
                                (float)(length *
random.NextDouble());

```

```

        //Compute a corresponding Y component that
will    // keep the same vector length.
        direction.Y = (float)Math.Sqrt(length*length
-
direction.X*direction.X);

        //Set the signs on the X and Y components to
match    // the signs from the original direction
vector.
        if(xNegative)
            direction.X = -direction.X;
        if(yNegative)
            direction.Y = -direction.Y;
        }//end NewDirection
        //-----
-----//

        public void Draw(SpriteBatch spriteBatch) {
            //Call the simplest available version of
            // spriteBatch.Draw

spriteBatch.Draw(image,position,Color.White);
        }//end Draw method
        //-----
-----//

        //This method is new to this version of the
Sprite    // class.
        //Returns the current rectangle occupied by
the    // sprite.
        public Rectangle GetRectangle() {
            return new Rectangle((int)(position.X),

```

```

        (int)(position.Y),
        image.Width,
        image.Height);
    } //end GetRectangle
    //-----
-----//

    //This method is new to this version of the
Sprite
    // class.
    //This method receives a list of Sprite
objects as
    // an incoming parameter. It tests for a
collision
    // with the sprites in the list beginning with
the
    // sprite at the head of the list. If it
detects a
    // collision, it stops testing immediately and
    // returns a reference to the Sprite object
for
    // which it found the collision. If it doesn't
find
    // a collision with any sprite in the list, it
    // returns null.
    //A collision is called if the rectangle
containing
    // this object's image intersects the
rectangle
    // containing a target sprite's image.
    public Sprite IsCollision(List<Sprite> target)
{
    Rectangle thisRectangle =
        new Rectangle((int)
(position.X),
                    (int)
(position.Y),

```

```

image.Width,
image.Height);
    Rectangle targetRectangle;
    int cnt = 0;

    while(cnt < target.Count){
        targetRectangle =
target[cnt].GetRectangle();

if(thisRectangle.Intersects(targetRectangle)){
    return target[cnt];
} //end if
    cnt++;
} //end while loop

    return null; //no collision detected
} //end IsCollision
//-----
-----//

    } //end class
} //end namespace

```

Note:

Listing 15 . The Game1 class for the project named XNA0130Proj.

```

/*Project XNA0130Proj
 * This project demonstrates how to integrate
 * spiders, and ladybugs in a program using
 * objects of a Sprite class with collision
 * detection.
 *

```



```

*****
***/
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using XNA0130Proj;

namespace XNA0130Proj {

    public class Game1 :
Microsoft.Xna.Framework.Game {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        //Use the following values to set the size of
the
        // client area of the game window. The actual
window
        // with its frame is somewhat larger depending
on
        // the OS display options. On my machine with
its
        // current display options, these dimensions
        // produce a 1024x768 game window.
        int windowWidth = 1017;
        int windowHeight = 738;

        //This is the length of the greatest distance
in
        // pixels that any sprite will move in a
single
        // frame of the game loop.
        double maxVectorLength = 5.0;

        Sprite spiderWeb;//reference to a background
sprite.

```

```

        //References to the spiders are stored in this
        // List object.
        List<Sprite> spiders = new List<Sprite>();
        int numSpiders = 200;//Number of spiders.
        //The following value should never exceed 60
moves
        // per second unless the default frame rate is
also
        // increased to more than 60 frames per
second.
        double maxSpiderSpeed = 30;//moves per second

        //References to the Ladybugs are stored in
this List.
        List<Sprite> ladybugs = new List<Sprite>();
        int numLadybugs = 5;//Max number of ladybugs
        double maxLadybugSpeed = 15;

        //Random number generator. It is best to use a
single
        // object of the Random class to avoid the
        // possibility of using different streams that
        // produce the same sequence of values.
        //Note that the random.NextDouble() method
produces
        // a pseudo-random value where the sequence of
values
        // is uniformly distributed between 0.0 and
1.0.
        Random random = new Random();
        //-----
-----//

        public Game1() { //constructor
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";

```

```

        //Set the size of the game window.
        graphics.PreferredBackBufferWidth =
windowWidth;
        graphics.PreferredBackBufferHeight =
windowHeight;
    }//end constructor
    //-----
-----//

    protected override void Initialize() {
        //No initialization required.
        base.Initialize();
    }//end Initialize
    //-----
-----//

    protected override void LoadContent() {
        spriteBatch = new
SpriteBatch(GraphicsDevice);

        //Create a sprite for the background image.
        spiderWeb =
            new
Sprite("spiderwebB",Content,random);
        spiderWeb.Position = new Vector2(0f,0f);

        //Instantiate all of the spiders and cause
them to
        // move from left to right, top to
        // bottom. Pass a reference to the same
Random
        // object to all of the sprites.
        for(int cnt = 0;cnt < numSpiders;cnt++) {
            spiders.Add(
                new

```

```

Sprite("blackWidowSpider",Content,random));

        //Set the position of the current spider
at a
        // random location within the game window.
        spiders[cnt].Position = new Vector2(
            (float)(windowWidth *
random.NextDouble()),
            (float)(windowHeight *
random.NextDouble()));

        //Get a direction vector for the current
spider.
        // Make both components positive to cause
the
        // vector to point down and to the right.
        spiders[cnt].Direction = DirectionVector(
            (float)maxVectorLength,
            (float)(maxVectorLength *
random.NextDouble()),
            false, //xNeg
            false); //yNeg

        //Notify the spider object of the size of
the
        // game window.
        spiders[cnt].WindowSize =
            new
Point(windowWidth,windowHeight);

        //Set the speed in moves per second for
the
        // current spider to a random value
between
        // maxSpiderSpeed/2 and maxSpiderSpeed.
        spiders[cnt].Speed = maxSpiderSpeed / 2
            + maxSpiderSpeed *

```

```

random.NextDouble() / 2;
    }//end for loop

    //Use the same process to instantiate all of
the
    // ladybugs and cause them to move from
right to
    // left, bottom to top.
    for(int cnt = 0;cnt < numLadybugs;cnt++) {
        ladybugs.Add(
            new
Sprite("ladybug",Content,random));
        ladybugs[cnt].Position = new Vector2(
            (float)(windowWidth *
random.NextDouble()),
            (float)(windowHeight *
random.NextDouble()));
        ladybugs[cnt].Direction = DirectionVector(
            (float)maxVectorLength,
            (float)(maxVectorLength *
random.NextDouble()),
            true,//xNeg
            true);//yNeg
        ladybugs[cnt].WindowSize =
            new
Point(windowWidth,windowHeight);
        ladybugs[cnt].Speed = maxLadybugSpeed / 2
            + maxLadybugSpeed *
random.NextDouble() / 2;
    }//end for loop

    }//end LoadContent
    //-----
    -----//

    //This method returns a direction vector given
the

```

```

        // length of the vector, the length of the
        // X component, the sign of the X component,
and the
        // sign of the Y component. Set negX and/or
negY to
        // true to cause them to be negative. By
adjusting
        // the signs on the X and Y components, the
vector
        // can be caused to point into any of the four
        // quadrants.
        private Vector2 DirectionVector(float vecLen,
                                         float xLen,
                                         Boolean negX,
                                         Boolean negY)
    {
        Vector2 result = new Vector2(xLen,0);
        result.Y = (float)Math.Sqrt(vecLen * vecLen
                                     - xLen *
xLen);
        if(negX)
            result.X = -result.X;
        if(negY)
            result.Y = -result.Y;
        return result;
    }//end DirectionVector
    //-----
-----//

    protected override void UnloadContent() {
        //No content unload required.
    }//end unloadContent
    //-----
-----//

    protected override void Update(GameTime
gameTime) {

```

```

        //Tell all the spiders in the list to move.
        for(int cnt = 0;cnt < spiders.Count;cnt++) {
            spiders[cnt].Move(gameTime);
        }//end for loop

        //Tell all the ladybugs in the list to move.
        for(int cnt = 0;cnt < ladybugs.Count;cnt++)
        {
            ladybugs[cnt].Move(gameTime);
        }//end for loop

        //Tell each ladybug to test for a collision
with a
        // spider and to return a reference to the
spider
        // if there is a collision. Return null if
there is
        // no collision.
        for(int cnt = 0;cnt < ladybugs.Count;cnt++)
        {

            //Test for a collision between this
ladybug and
            // all of the spiders in the list of
spiders.
            Sprite target =

ladybugs[cnt].IsCollision(spiders);
            if(target != null) {
                //There was a collision. Cause the
spider to
                // move 128 pixels to the right.
                target.Position =
                    new Vector2(target.Position.X
+ 128,
target.Position.Y);

```

```

        //If the collision was with a black
widow
        // spider, cause it to reincarnate into
a
        // green spider.
        if(target.Image.Name ==
"blackWidowSpider") {
target.SetImage("greenspider",Content);
        }//If the collision was with a green
spider,
        // cause it to reincarnate into a brown
        // spider.
        else if(target.Image.Name ==
"greenspider") {
target.SetImage("brownSpider",Content);
        }//If the collision was with a brown
spider,
        // it gets eaten. Remove it from the
list of
        // spiders.
        else if(target.Image.Name ==
"brownSpider") {
            spiders.Remove(target);
        }// end else-if
        }//end if
    }//end for loop

    base.Update(gameTime);
} //end Update method
//-----
-----//

protected override void Draw(GameTime
gameTime) {

```



```

        spriteBatch.Begin();

        spiderWeb.Draw(spriteBatch);//draw
background

        //Draw all spiders.
        for(int cnt = 0;cnt < spiders.Count;cnt++) {
            spiders[cnt].Draw(spriteBatch);
        }//end for loop

        //Draw all ladybugs.
        for(int cnt = 0;cnt < ladybugs.Count;cnt++)
{
            ladybugs[cnt].Draw(spriteBatch);
        }//end for loop

        spriteBatch.End();

        base.Draw(gameTime);
    }//end Draw method
    //-----
-----//
} //end class
} //end namespace

```

-end-

Xna0132-A Simple Game Program with On-Screen Text

Learn how to write a simple game program involving user input from the keyboard and the mouse. Also learn how to create on-screen text.

Revised: Tue May 10 11:37:12 CDT 2016

This page is part of a Book titled [XNA Game Studio](#).

Table of Contents

- [Table of Contents](#)
- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Preview](#)
 - [Spider behavior](#)
 - [The ladybugs](#)
 - [Using the keyboard](#)
 - [Using the mouse](#)
 - [Difficulty level](#)
 - [A useful student project](#)
 - [Alternatives to the arrow keys](#)
- [Discussion and sample code](#)
 - [How to draw on-screen text](#)
 - [Font availability](#)
 - [Identifying the font families](#)
 - [Matching font families to font files](#)

- [Getting ready to draw on-screen text](#)
- [Drawing the on-screen text](#)
- [The project named XNA0132ProjA](#)
 - [The overridden Game.Draw method](#)
 - [The SpriteBatch.DrawString method](#)
 - [Purpose of the origin parameter](#)
 - [All I need to say](#)
- [The Sprite class for XNA0132Proj](#)
 - [The new code](#)
 - [A new read-only_property_accessor method for the Edge property](#)
 - [New version of the Move method](#)
 - [End of discussion of the Sprite class](#)
- [The Game1 class for XNA0132Proj](#)
 - [General information](#)
 - [The objective of the game](#)
 - [Ladybug control](#)
 - [Spiders are recycled](#)
 - [Keeping score](#)
 - [Code for the Game1 Class](#)
 - [The overridden LoadContent method](#)
 - [The font family](#)
 - [Make the mouse pointer visible](#)
 - [Position the on-screen text](#)
 - [Set the initial position of the mouse pointer](#)
 - [End of the LoadContent method](#)
 - [The overridden Update method](#)

- [Moving the ladybug objects using the keyboard](#)
- [Dragging the ladybug objects with the mouse](#)
- [The overridden Game.Draw method](#)
 - [The MeasureString method](#)
 - [Overloaded division operator](#)
- [The end of the program](#)
 - [Run the program](#)
 - [Run my text program](#)
 - [Run my_game program](#)
 - [Summary](#)
 - [What's next?](#)
 - [Miscellaneous](#)
 - [Complete program listing](#)

Preface

This module is one in a collection of modules designed primarily for teaching **GAME 1343 Game and Simulation Programming I** at Austin Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

An earlier module titled [Getting Started](#) provided information on how to get started programming with Microsoft's XNA Game Studio.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Reduced view of the game window.
- [Figure 2](#). Information on fonts from the XNA documentation.
- [Figure 3](#). Correspondence between font-family names and font file names.
- [Figure 4](#). Demonstration of on-screen text with the Lindsey font.

Listings

- [Listing 1](#). Beginning of the Game1 class and the overridden LoadContent method for the project named XNA0132ProjA.
- [Listing 2](#). Beginning of the overridden Game.Draw method.
- [Listing 3](#). Remainder of the overridden Game.Draw method.
- [Listing 4](#). A new read-only property accessor method for the Edge property for the Sprite class.
- [Listing 5](#). New version of the Move method.
- [Listing 6](#). Abbreviated beginning of the Game1 Class for the XNA0132Proj project.
- [Listing 7](#). Abbreviated overridden LoadContent method.
- [Listing 8](#). Beginning of an abbreviated overridden Update method.
- [Listing 9](#). Service the keyboard.
- [Listing 10](#). Moving the ladybug objects with the mouse.
- [Listing 11](#). The overridden Game.Draw method.
- [Listing 12](#). Game1 class for project XNA0132ProjA.
- [Listing 13](#). Sprite class for project XNA0132Proj.
- [Listing 14](#). Game1 class for project XNA0132Proj.

General background information

When we finished the previous module, we were missing one critical element needed to write a simple game program: ***user input***. We were also lacking some other tools that would make it possible for us to create more interesting 2D arcade-style games including the ability to create on-screen text.

In this module, you will learn how to program for user input with the keyboard and the mouse. You will also learn how to create on-screen text.

Preview

The program that I will explain in this module is a simple game in which ten spiders attempt to traverse their web from top to bottom. Two large ladybugs are there to eat them if possible. (See [Figure 1](#).) The objective of the game is to prevent as many of the spiders as possible from making it to the bottom of the game window.

[Figure 1](#) shows a reduced view of the game window shortly after the game is started and before any spiders make it to the bottom.

Note:

Figure 1 . Reduced view of the game window.



Spider behavior

The spiders move at different speeds in different directions but generally towards the southeast. If a spider makes it to the bottom, it is counted as having crossed the finish line. That spider wraps back up to the top and starts the journey again. If a spider hits the right edge of the window, it wraps around to the left edge and continues the journey.

The ladybugs

The movements of the ladybugs are under the control of the player. If the player causes a ladybug to collide with a spider, the spider is eaten and removed from the spider population.

Using the keyboard

The player can move the ladybugs using either the keyboard or the mouse. Pressing the four arrow keys causes one of the ladybugs to move in the corresponding directions. Pressing two arrow keys at the same time causes the ladybug to move in a diagonal direction.

Holding down the A key and pressing the arrow keys causes the other ladybug to move.

Using the mouse

Pressing the left mouse button makes it possible to drag a ladybug with the mouse. Pressing the right mouse button makes it possible to drag the other ladybug with the mouse. Pressing both mouse buttons makes it possible to drag both ladybugs with the mouse.

Difficulty level

The game has only one level. I personally find it very difficult to prevent all ten spiders from making it to the bottom using the arrow keys only. On the other hand, it is easy to defeat the spiders by dragging a ladybug with the mouse.

A useful student project

A useful student project would be to upgrade the program to provide varying levels of difficulty. This could be accomplished by changing the speed of the spiders, changing the number of spiders, or a combination of the two.

Alternatives to the arrow keys

Another useful student project would be to define alternatives to the arrow keys, such as the keys labeled 4, 8, 6, and 2 on the keypad, or four keys in a diamond pattern on the left side of the keyboard.

Discussion and sample code

As usual, I will break this program down and explain it in fragments. However, I will only explain those portions of the code that are substantially different from code that I have explained in earlier modules.

Complete listings of the **Sprite** class and the **Game1** class are provided in [Listing 13](#) and [Listing 14](#) near the end of the module.

Before getting into the game program, however, I will explain a very short program that illustrates how to draw on-screen text. I personally found the [documentation](#) on this topic to be somewhat confusing.

How to draw on-screen text

The difficulty that I encountered in drawing on-screen text was not in the writing of program code. Rather, it was in getting ready to write program code.

Font availability

Before getting into the details, I want to mention that fonts are typically copyrighted items of intellectual property and you must be careful when distributing fonts for which you don't have distribution rights. To assist you in this regard, the information shown in [Figure 2](#) currently appears in the [documentation](#).

Note:

Figure 2 . Information on fonts from the XNA documentation.
The following list of fonts are installed by XNA Game Studio and are redistributable:

- Kooten.ttf
- Linds.ttf
- Miramo.ttf
- Bold Miramob.ttf
- Peric.ttf
- Pericl.ttf
- Pesca.ttf
- Pescab.ttf

These OpenType fonts, created by Ascender Corporation and licensed by Microsoft, are free for you to use in your XNA Game Studio game. You may redistribute these fonts in their original format as part of your game. These fonts were previously available only on the XNA Creators Club Online website.

Identifying the font families

You should be able to select **Fonts** in the Windows **Control Panel** and see a list of the font families that are installed on your machine. The problem is that the names of the font families are not an exact match for the file names listed in [Figure 2](#).

Matching font families to font files

[Figure 3](#) shows my best guess as to the correspondence between font-family names and the file names listed in [Figure 2](#). Font family names are on the left and font file names are on the right.

Note:

Figure 3 . Correspondence between font-family names and font file names.

- Kootenay - Kooten.ttf
- Lindsey - Linds.ttf
- Miramonte - Miramo.ttf
- Miramonte Bold - Bold Miramob.ttf
- Pericles - Peric.ttf
- Pericles Light - Pericl.ttf
- Pescadero - Pesca.ttf
- Pescadero Bold -Pescab.ttf

You will learn shortly why we care about the font-family names.

Getting ready to draw on-screen text

Before you can draw on-screen text you must add a **Sprite Font** to your project from the IDE during the design process.

Here is my interpretation of the steps required to add the **Sprite Font** to your project:

- Open your project in Visual C#.
- Right-click your **Content** folder in **Solution Explorer** , select **Add** , and then click **New Item** .
- From the **Add New Item** dialog box, click **Sprite Font** .
- Change the text in the **Name** field to **[Font-Family Name].spritefont** , such as **Lindsey.spritefont** for example.
- Click the **Add** button. This will create a new XML file named **[Font-Family Name].spritefont** , which will be opened in the IDE editor.
- Click the file named **[Font-Family Name].spritefont** in the **Solution Explorer** and note the **Asset Name** that is displayed in the **Properties** window (such as **Lindsey** for example).
- The XML file contains an element named **FontName** . Change the contents of that element, if necessary, to match the name of the font family that you specified.
- Change the contents of the **Size** element in the XML file to the point size you desire for your font.
- Change the contents of the **Style** element in the XML file to the style of font to import. You can specify **Regular** , **Bold** , **Italic** , or **Bold, Italic** . Note that the contents of the XML elements are case sensitive.
- Specify the character regions to import for this font. Character regions specify which characters in the font are rendered by the **SpriteFont** . You can specify the start and end of the region by using the characters themselves, or by using their decimal values with an ampersand-pound sign prefix. The default character region includes all the characters between the space and tilde characters, inclusive.
- Follow the instructions in the comments in the XML file to make any other changes to the contents of the XML elements that are appropriate for your program.

Drawing the on-screen text

I will explain the code in the **Game1** class of the project named **XNA0132ProjA** that produced the screen output shown in reduced form in

[Figure 4.](#)

Note:

Figure 4 . Demonstration of on-screen text with the Lindsey font.



As usual, I will explain the code in fragments. A complete listing of the **Game1** class is provided in [Listing 12](#) near the end of the module.

Here is my interpretation of the coding steps for displaying on-screen text in the game window.

- Add a **Sprite Font** to your project as described above.
- Declare an instance variable of type [SpriteFont](#) to refer to a **SpriteFont** object. (See [Listing 1.](#))
- Declare an instance variable of type **Vector2** to refer to an object that specifies the position of your text relative to the upper left corner of the game window. (See [Listing 1.](#))

- Create a **SpriteFont** object by calling the **ContentManager.Load** method, specifying the **SpriteFont** class and the **Asset Name** of the imported font in your overridden **LoadContent** method. Save the object's reference in the instance variable declared earlier. (See [Listing 1](#).)
- Instantiate a new **Vector2** object to specify the position of the text in the game window in the **LoadContent** method. Save the object's reference in the instance variable declared earlier. (See [Listing 1](#).) The X and Y component values of the **Vector2** object can be modified later in the **Update** method if needed.
- Write the typical **SpriteBatch.Begin** and **SpriteBatch.End** method calls in the overridden **Game.Draw** method. (See [Listing 2](#) and [Listing 3](#).)
- Create a string in the **LoadContent** , **Update** , or **Draw** methods containing the text that will be displayed in the game window. (See [Listing 2](#).)
- Instantiate a **Vector2** object in the **LoadContent** , **Update** , or **Draw** methods that specifies the origin of the text string, which will be drawn at the previously computed position of the text string. (See [Listing 2](#).)
- Compute values for the other parameters to the **SpriteBatch.DrawString** method as needed.
- Call the **SpriteBatch.DrawString** method between the **SpriteBatch.Begin** and **SpriteBatch.End** methods. (See [Listing 3](#).)

The project named **XNA0132ProjA**

[Listing 1](#) shows the beginning of the **Game1** class and the overridden **LoadContent** method for the project named **XNA0132ProjA** . This project, which is different from **XNA0132Proj** to be discussed later, demonstrates how to draw onscreen text.

Note:

Listing 1 . Beginning of the **Game1** class and the overridden **LoadContent** method for the project named **XNA0132ProjA**.

```

namespace XNA0132ProjA {

    public class Game1 :
Microsoft.Xna.Framework.Game {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        SpriteFont Font1;
        Vector2 FontPos;
        //-----
        -----//

        protected override void LoadContent() {
            spriteBatch = new
SpriteBatch(GraphicsDevice);

            //Create a new SpriteFont object.
            Font1 = Content.Load<SpriteFont>("Lindsey");

            //Create a new Vector2 object to center the
text
            // in the game window.
            FontPos = new Vector2(
                graphics.GraphicsDevice.Viewport.Width
/ 2,
graphics.GraphicsDevice.Viewport.Height / 2);
        } //end LoadContent method

```

There is no unusual code in [Listing 1](#). However, if my memory serves me correctly, this is the first time in this series of modules that I have called the **Content.Load** method to load a resource other than an image.

The overridden Game.Draw method

The **Game.Draw** method begins in [Listing 2](#).

Note:

Listing 2 . Beginning of the overridden Game.Draw method.

```
protected override void Draw(GameTime
gameTime) {
    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin();

    // Draw Lindsey Font
    string output = "Lindsey Font";

    // Find the center of the string
    Vector2 FontOrigin =
Font1.MeasureString(output) / 2;
```

There is nothing new or unusual in [Listing 2](#).

The remainder of the **Game.Draw** method is shown in [Listing 3](#).

Note:

Listing 3 . Remainder of the overridden Game.Draw method.

```
//Draw the string with the FontOrigin at the
// position specified by FontPos.

spriteBatch.DrawString(
    Font1, //font to use
    output, //output text
```

```
        FontPos,//position re upper left corner
        Color.LightGreen,//color of text
        -45,//rotation of text
        FontOrigin,//place this at FontPos
        2.5f,//scale
        SpriteEffects.None,
        0.5f);//z-order layer

    spriteBatch.End();

    base.Draw(gameTime);
} //end Draw method
```

There is nothing unusual in the code in [Listing 3](#). However, this is the first time in this series of modules that I have called the **SpriteBatch.DrawString** method.

The **SpriteBatch.DrawString** method

There are several overloaded versions of the **SpriteBatch.DrawString** method. The different versions depend on the features provided by the parameters. The version called in [Listing 3](#) is one of the most complex. Here is part of what the [documentation](#) has to say about this version of the **DrawString** method:

"Adds a mutable sprite string to the batch of sprites to be rendered, specifying the font, output text, screen position, color tint, rotation, origin, scale, and effects."

Parameters

- **spriteFont** - The sprite font.
- **text** - The mutable (read/write) string to draw.

- **position** - The location, in screen coordinates, where the text will be drawn.
- **color** - The desired color of the text.
- **rotation** - The angle, in radians, to rotate the text around the origin.
- **origin** - The origin of the string. Specify (0,0) for the upper-left corner.
- **scale** - Uniform multiple by which to scale the sprite width and height.
- **effects** - Rotations to apply prior to rendering.
- **layerDepth** - The sorting depth of the sprite, between 0 (front) and 1 (back)."

Purpose of the origin parameter

Everything in this description is reasonably clear except for the description of the **origin** parameter so I will attempt to clarify it. The origin specifies a point within the text string relative to its upper left corner that will be drawn at the point in the game window specified by the **position** parameter.

All I need to say

That is probably all I need to say about the on-screen display of text. It's time to move on to an explanation of our simple game program named **XNA0132Proj** beginning with the **Sprite** class.

The **Sprite** class for **XNA0132Proj**

The **Sprite** class that I used in this program is very similar to the class that I explained in the previous module. Therefore, I won't have a lot to say about the **Sprite** class in this module. A complete listing of the **Sprite** class is provided in [Listing 13](#) near the end of the module.

This version of the **Sprite** class supports collision detection based on intersecting rectangles. It also provides a new **Edge** property that records and returns the edge number (1, 2, 3, or 4) if a sprite collides with an edge of the game window. However, the edge information is available for only

one iteration of the game loop before it changes back to the normal value of 0.

The value of the **Edge** property temporarily changes to 1, 2, 3, or 4 if a sprite collides with the top, right, bottom, or left edge of the game window respectively.

Since the information regarding the collision with an edge is the only change to this **Sprite** class relative to the version that I explained in a previous module, that is all I will discuss about the **Sprite** class in this module.

The new code

The first manifestation of change in this version of the **Sprite** class is the simple declaration of an instance variable of type **int** named **edge** near the beginning of the class definition. You can view that statement in [Listing 13](#).

A new read-only property accessor method for the Edge property

The next manifestation is the read-only property accessor method for the new **Edge** property shown in [Listing 4](#).

Note:

Listing 4 . A new read-only property accessor method for the Edge property for the Sprite class.

```
public int Edge {  
    get {  
        return edge;  
    } //end get  
} //end Edge property accessor
```

New version of the Move method

[Listing 5](#) shows the new version of the **Move** method.

Note:

Listing 5 . New version of the Move method.

```
//This method causes the sprite to move in the
// direction of the direction vector if the
elapsed
// time since the last move exceeds the
elapsed
// time target based on the specified speed.
public void Move(GameTime gameTime) {

    //New to this version
    //Clear the Edge property value. Edge
information
    // is available for only one iteration of
the
    // game loop.
    edge = 0;

    //Accumulate elapsed time since the last
move.
    elapsedTime +=
gameTime.ElapsedGameTime.Milliseconds;

    if(elapsedTime > elapsedTimeTarget) {
        //It's time to make a move. Set the
elapsed
        // time to a value that will attempt to
produce
        // the specified speed on the average.
        elapsedTime -= elapsedTimeTarget;
```

```

        //Add the direction vector to the position
        // vector to get a new position vector.
        position =
Vector2.Add(position,direction);

        //Check for a collision with an edge of
the game
        // window. If the sprite reaches an edge,
cause
        // the sprite to wrap around and reappear
at the
        // other edge, moving at the same speed in
a
        // different direction within the same
quadrant
        // as before. Also set the Edge property
to
        // indicate which edge was involved. 1 is
top, 2
        // is right, 3 is bottom, and 4 is left.
        // Note that the Edge property will be
cleared
        // to 0 the next time the Move method is
called.
        if(position.X < -image.Width) {
            position.X = windowSize.X;
            edge = 4;//collision with the left edge
- new
            NewDirection();
        }//end if

        if(position.X > windowSize.X) {
            position.X = -image.Width / 2;
            edge = 2;//collision with the right edge
- new
            NewDirection();

```

```

        }//end if

        if(position.Y < -image.Height) {
            position.Y = windowSize.Y;
            edge = 1;//collision with the top - new
            NewDirection();
        }//end if

        if(position.Y > windowSize.Y) {
            position.Y = -image.Height / 2;
            edge = 3;//collision with the bottom -
new
            NewDirection();
        }//end if on position.Y
    }//end if on elapsed time
} //end Move

```

There is nothing unusual about the code that was added to the **Move** method, so it shouldn't require further explanation beyond the embedded comments.

End of discussion of the Sprite class

There were no additional changes made to the **Sprite** class other than those identified above, so that will end the discussion of the **Sprite** class.

The Game1 class for XNA0132Proj

General information

This project demonstrates how to write a simple 2D arcade style game.

Ten spiders try to make it from the top to the bottom of a spider web.

The bottom of the web is guarded by two large ladybugs that are intent on eating the spiders.

If a ladybug collides with a spider, the spider gets eaten by the ladybug and removed from the population of spiders. If the player plays long enough, all of the spiders should eventually be eaten.

The objective of the game

The objective of the game is for the player to cause as many spiders as possible to be eaten by the ladybugs before they make it to the bottom of the game window.

Ladybug control

The ladybugs can be moved by the player using either the keyboard or the mouse. Pressing the arrow keys moves one of the ladybugs. Holding down the A key and pressing the arrow keys moves the other ladybug.

The player can drag one of the ladybugs with the mouse pointer by pressing the left mouse button. The player can drag the other ladybug by pressing the right mouse button.

Spiders are recycled

Even though the game begins with only ten spiders, those that are not eaten before they reach the bottom of the game window are recycled. If a spider makes it from the top to the bottom of the game window without being eaten, the spider wraps around and appears back at the top of the game window ready to start the trip over.

Keeping score

A counter keeps track of the number of spiders that cross the bottom of the game window. On-screen text displays that number as the game progresses as shown in [Figure 1](#). Note, however, that if two or more spiders cross the bottom edge of the game window during the same iteration of the game loop, only one will be counted. In hindsight, therefore, the elapsed time required to cause the ladybugs to eat all of the spiders might be a better scorekeeping mechanism than counting spider crossings.

Modifying the program to keep score on the basis of elapsed time might make a good student project.

Code for the Game1 Class

[Listing 6](#) shows the beginning of the **Game1** class along with the constructor in abbreviated form. By abbreviated form, I mean that much of the code is very similar to code that I have explained in earlier modules so I deleted that code from [Listing 6](#) for brevity. You can view all of the code for the **Game1** class in [Listing 14](#).

Note:

Listing 6 . Abbreviated beginning of the Game1 Class for the XNA0132Proj project.

```
namespace XNA0132Proj {  
  
    public class Game1 :  
Microsoft.Xna.Framework.Game {  
        GraphicsDeviceManager graphics;  
        SpriteBatch spriteBatch;  
  
        //code deleted for brevity  
  
        //The following variable is used to count the
```

```

number
    // of spiders that make it past the ladybugs
and
    // reach the bottom of the game window.
    int spiderCount = 0;

    SpriteFont Font1;//font for on-screen text
    Vector2 FontPos;//position of on-screen text
    //-----
    -----//

    public Game1() { //constructor
        //code deleted for brevity
    } //end constructor

```

The variable declarations near the center of [Listing 6](#) are new to this module.

The overridden LoadContent method

An abbreviated listing of the **LoadContent** method is shown in [Listing 7](#).

Note:

Listing 7 . Abbreviated overridden LoadContent method.

```

    protected override void Update(GameTime
gameTime) {
        spriteBatch = new
SpriteBatch(GraphicsDevice);
        Font1 = Content.Load<SpriteFont>
("Kootenay");
        IsMouseVisible = true;//make mouse visible

```



```
//code deleted for brevity

//Position the on-screen text.
FontPos = new Vector2(windowWidth / 2, 50);

//Position the mouse pointer in the center
of the
// game window.
Mouse.SetPosition(windowWidth /
2, windowHeight / 2);

} //end LoadContent
```

The font family

To begin with, note that this program displays text on the screen using the font family named **Kootenay** (see [Figure 3](#) for a reference to this font-family name).

Make the mouse pointer visible

By default, the mouse pointer is not visible when it is in the game window. The code in [Listing 7](#) sets the inherited variable named **IsMouseVisible** to true, which causes the mouse pointer to be visible.

Position the on-screen text

The code in [Listing 7](#) causes the text to be centered horizontally, 50 pixels down from the top of the game window as shown in [Figure 1](#).

Set the initial position of the mouse pointer

The code in [Listing 7](#) causes the initial position of the mouse pointer to be centered in the game window. (I will have more to say about this later.)

End of the LoadContent method

[Listing 7](#) also signals the end of the overridden **LoadContent** method.

The overridden Update method

An abbreviated listing of the overridden **Update** method begins in [Listing 8](#)

.

Note:

Listing 8 . Beginning of an abbreviated overridden Update method.

```
protected override void Update(GameTime
gameTime) {

    //code deleted for brevity

    //Check to see if any spiders have made it
to the
    // bottom edge.
    for(int cnt = 0;cnt < spiders.Count;cnt++) {
        if(spiders[cnt].Edge == 3)
            //One or more made it to the bottom
edge.
            spiderCount += 1;
    }//end for loop
```

The code in [Listing 8](#) counts the number of spiders that make it to the bottom and collide with the bottom edge of the game window.

Moving the ladybug objects using the keyboard

The code required to service the keyboard is long, tedious, and repetitive, but is not complicated. That code is shown in [Listing 9](#).

Note:

Listing 9 . Service the keyboard.

```
//The following code is used to move one or
the
// other of two ladybugs using the arrow
keys.
// Press only the arrow keys to move one of
the
// ladybugs. Press the A plus the arrow keys
// to move the other ladybug.
//The ladybugs cannot be moved outside the
game
// window using the keyboard.
//While an arrow key is pressed, the ladybug
moves
//five pixels per call to the Update method.

//Get the state of the keyboard.
KeyboardState keyboardState =
Keyboard.GetState();

//Execute moves on one ladybug with arrow
keys plus
// the A or a key.
if(keyboardState.IsKeyDown(Keys.Left) &&
```

```

        (keyboardState.IsKeyDown(Keys.A)) &&
        (ladybugs[0].Position.X > 0)) {
            ladybugs[0].Position = new Vector2(
ladybugs[0].Position.X - 5,
ladybugs[0].Position.Y);
        }//end if

        if(keyboardState.IsKeyDown(Keys.Right) &&
            (keyboardState.IsKeyDown(Keys.A)) &&
            (ladybugs[0].Position.X <
                (windowWidth -
ladybugs[1].Image.Width))) {
            ladybugs[0].Position = new Vector2(
ladybugs[0].Position.X + 5,
ladybugs[0].Position.Y);
        }//end if

        if(keyboardState.IsKeyDown(Keys.Up) &&
            (keyboardState.IsKeyDown(Keys.A)) &&
            (ladybugs[0].Position.Y > 0)) {
            ladybugs[0].Position = new Vector2(
ladybugs[0].Position.X,
ladybugs[0].Position.Y - 5);
        }//end if

        if(keyboardState.IsKeyDown(Keys.Down) &&
            (keyboardState.IsKeyDown(Keys.A)) &&
            (ladybugs[0].Position.Y <
                (windowHeight -
ladybugs[1].Image.Height))) {
            ladybugs[0].Position = new Vector2(

```

```

ladybugs[0].Position.X,
ladybugs[0].Position.Y + 5);
    }//end if

    //Execute moves on the other ladybug with
arrow
    // keys pressed but the A key not pressed.
    if(keyboardState.IsKeyDown(Keys.Left) &&
        !(keyboardState.IsKeyDown(Keys.A)) &&
        (ladybugs[1].Position.X > 0)) {
        ladybugs[1].Position = new Vector2(
ladybugs[1].Position.X - 5,
ladybugs[1].Position.Y);
    }//end if

    if(keyboardState.IsKeyDown(Keys.Right) &&
        !(keyboardState.IsKeyDown(Keys.A)) &&
        (ladybugs[1].Position.X <
            (windowWidth -
ladybugs[1].Image.Width))) {
        ladybugs[1].Position = new Vector2(
ladybugs[1].Position.X + 5,
ladybugs[1].Position.Y);
    }//end if

    if(keyboardState.IsKeyDown(Keys.Up) &&
        !(keyboardState.IsKeyDown(Keys.A)) &&
        (ladybugs[1].Position.Y > 0)) {
        ladybugs[1].Position = new Vector2(
ladybugs[1].Position.X,

```

```

ladybugs[1].Position.Y - 5);
    }//end if

    if(keyboardState.IsKeyDown(Keys.Down) &&
        !(keyboardState.IsKeyDown(Keys.A)) &&
        (ladybugs[1].Position.Y <
            (windowHeight -
ladybugs[1].Image.Height))) {
        ladybugs[1].Position = new Vector2(
ladybugs[1].Position.X,
ladybugs[1].Position.Y + 5);
    }//end if

```

Get the state of the keyboard

The code [Listing 9](#) calls the static **GetState** method of the **Keyboard** class to get a reference to an object of the **KeyboardState** structure. Here is part of what the [documentation](#) for **KeyboardState** has to say:

Note: "Represents a state of keystrokes recorded by a keyboard input device."

Polling instead of handling events

Most business and scientific programs written using C# would sit idle and wait for a keyboard event to occur. Then the event would be handled, and the program would sit idle and wait for the next event to occur.

Game programs written using XNA don't sit idle. Such programs are always executing the game loop. Therefore, XNA game programs poll the keyboard and the mouse instead of waiting to handle events fired by the keyboard or the mouse. This represents a major difference in programming philosophy.

Key up or down queries

An object of the **KeyboardState** structure has several methods, including the following two which are of particular interest:

- [IsKeyDown](#) - Returns whether a specified key is currently being pressed.
- [IsKeyUp](#) - Returns whether a specified key is currently not pressed.

In this program, our primary interest is in the first of the two methods.

An incoming parameter of type Keys

Both methods require an incoming parameter of type [Keys](#), which is an *"Enumerated value that specifies the key to query."*

The **Keys** Enumeration has many members, each one of which *"Identifies a particular key on a keyboard."* In this program, we are interested in the following enumerated values:

- **Down** - DOWN ARROW key
- **Left** - LEFT ARROW key
- **Right** - RIGHT ARROW key
- **Up** - UP ARROW key
- **A** - The A key

These are the enumerated values that we will use to determine if the player is pressing one or more arrow keys and the A key.

Make the test

The code in [Listing 9](#) tests to determine if the player is currently pressing both the LEFT ARROW key and the A key. In addition the code tests to

confirm that the current position of the ladybug is not outside the left edge of the game window.

Move the ladybug

If this test returns true, the code in [Listing 9](#) sets the horizontal position for the ladybug to five pixels to the left of its current position. This change in position will be reflected visually in the game window the next time the ladybug is drawn.

Three more tests

Three more similar **if** statements are executed testing for RIGHT ARROW, UP ARROW, and DOWN ARROW and moving the ladybug appropriately if the tests return true.

Not mutually exclusive

Note that these tests are not mutually exclusive. For example, the player can be pressing both the LEFT ARROW key and the UP ARROW key, which will result in the ladybug moving diagonally up and to the left.

Applies to a particular ladybug

The tests performed above and the corresponding moves apply only to the ladybug whose reference is stored at index [1] in the list of ladybugs.

Four more similar tests

Four more very similar tests are applied by the code in [Listing 9](#). These tests and the resulting moves apply only to the ladybug whose reference is stored at index [0] in the list of ladybugs.

Testing for the A key not pressed

Note that in these four tests, I used the logical **not** operator (!) to test for the arrow keys pressed but the A key **not pressed**. (I also could have called the **IsKeyUp** method instead of using the logical not operator for these tests.)

[Listing 10](#) shows the code required to drag the ladybug objects with the mouse.

Note:

Listing 10 . Moving the ladybug objects with the mouse.

```
//Get the state of the mouse.
MouseState mouseState = Mouse.GetState();
//Press the left mouse button to move one
ladybug.
    if(mouseState.LeftButton ==
ButtonState.Pressed) {
        ladybugs[0].Position =
            new
Vector2(mouseState.X,mouseState.Y);
    }//end if

    //Press the right mouse button to move the
other
    // ladybug.
    if(mouseState.RightButton ==
ButtonState.Pressed) {
        ladybugs[1].Position =
            new
Vector2(mouseState.X,mouseState.Y);
    }//end if

    base.Update(gameTime);
} //end Update method
```

Relatively simple code

The code in [Listing 10](#) is used to drag one or the other of two ladybug objects with the mouse. As you can see from the amount of code involved, dragging the ladybug objects with the mouse is much simpler than moving them from the keyboard.

You can drag one of the ladybug objects by pressing the left mouse button and you can drag the other ladybug object by pressing the right mouse button. You can drag them both, one on top of the other by pressing both mouse buttons at the same time.

Get the state of the mouse

As is the case with the keyboard, the code in [Listing 10](#) polls the mouse once during each iteration of the game loop to determine its state.

[Listing 10](#) begins by calling the **GetState** method of the **Mouse** class to get a reference to a **MouseState** object that describes the current state of the mouse.

A MouseState object

According to the [documentation](#), a **MouseState** object

Note: "Represents the state of a mouse input device, including mouse cursor position and buttons pressed."

The object contains several properties including the following:

- **LeftButton** - Returns the state of the left mouse button as type **ButtonState** .
- **RightButton** - Returns the state of the right mouse button as type **ButtonState** .

- **X** - Specifies the horizontal position of the mouse cursor in pixels relative to the upper left corner of the game window.
- **Y** - Specifies the vertical position of the mouse cursor in pixels relative to the upper left corner of the game window.

The **ButtonState** enumeration

According to the [documentation](#), **ButtonState** is an Enumeration that

Note: "Identifies the state of a mouse button or Xbox 360 Controller button."

It has the following members:

- **Pressed** - The button is pressed.
- **Released** - The button is released.

Test for a left mouse button pressed

The code in [Listing 10](#) tests to determine if the left mouse button is pressed. If so, it sets the position of the ladybug object referred to by index [0] in the list of ladybugs to the current position of the mouse pointer.

A change in ladybug position

This change in position becomes visible the next time the ladybug is drawn. This has the effect of causing the ladybug object to follow the mouse pointer throughout the game window. It is even possible for the mouse to drag a ladybug outside the game window and leave it there.

Do the same for the other ladybug object

The code in [Listing 10](#) also does essentially the same thing, but applies the operation to the ladybug object referred to by index [1] in the list of ladybugs.

Methods of the Mouse class

In addition to the **GetState** method mentioned earlier, the **Mouse** class has one other static method that is not inherited from the **Object** class. It is named [SetPosition](#), and it

Note: "Sets the position of the mouse cursor relative to the upper-left corner of the window."

You saw this method used to place the mouse cursor in the center of the game window in [Listing 7](#).

The end of the overridden Update method

[Listing 10](#) also signals the end of the overridden **Update** method.

The overridden **Game.Draw** method

The overridden **Game.Draw** method is shown in its entirety in [Listing 11](#).

Note:

Listing 11 . The overridden **Game.Draw** method.

```
protected override void Draw(GameTime
gameTime) {

    spriteBatch.Begin();

    spiderWeb.Draw(spriteBatch); //draw
background
```

```

        //Draw all spiders.
        for(int cnt = 0;cnt < spiders.Count;cnt++) {
            spiders[cnt].Draw(spriteBatch);
        }//end for loop

        //Draw all ladybugs.
        for(int cnt = 0;cnt < ladybugs.Count;cnt++)
        {
            ladybugs[cnt].Draw(spriteBatch);
        }//end for loop

        //Draw the output text.
        string output = "";
        if(spiderCount == 0){
            output = "Congratulations. No spiders made
it to"
                    + " the bottom.";
        }else{
            output = "Oops, " + spiderCount + " or
more "
                    + "spiders made it to the
bottom.";
        }//end else

        // Find the center of the string
        Vector2 FontOrigin =
Font1.MeasureString(output) / 2;
        // Draw the string
        spriteBatch.DrawString(Font1,
                                output,
                                FontPos,
                                Color.Yellow,
                                0,//angle
                                FontOrigin,
                                1.0f,//scale
                                SpriteEffects.None,

```

```

0.0f); //layer depth

spriteBatch.End();

base.Draw(gameTime);
} //end Draw method
//-----
-----//
} //end class
} //end namespace

```

The MeasureString method

Other than the [MeasureString](#) method of the [SpriteFont](#) class, there is nothing in [Listing 11](#) that I haven't explained before.

The MeasureString method

Note: "Returns the height and width of a given string as a Vector2."

Overloaded division operator

The statement that declares and initializes the variable named **FontOrigin** in [Listing 11](#) is very interesting. Recall that a **Vector2** object has two fields, X and Y. Note that the code in [Listing 11](#) divides the object by 2. I didn't say that the code extracts the fields and divides them by 2. I said that the code divides the object by 2. This must mean that the division operator (/) is overloaded by the **Vector2** class such that dividing a **Vector2** object by a scalar value divides the fields by the scalar value.

If you know what I am talking about when I talk about operator overloading, that is good. If not, don't worry about it. That topic is far

beyond the technical requirements for the introductory XNA game programming course that I teach.

The end of the program

[Listing 11](#) signals the end of the overridden **Game.Draw** method, the end of the **Game1** class, and the end of the program.

Run the program

I encourage you to copy the code from [Listing 13](#) and [Listing 14](#). Use that code to create an XNA project. You should be able to download suitable images from the Internet. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Run my text program

Click [here](#) to download a zip file containing my version of the program named **XNA0132ProjA**. Extract the folder named **XNA0132ProjA** from the zip file and save it somewhere on your disk. Start **Visual C# 2010 Express** and select **Open Project...** from the **File** menu. Navigate to the project folder and select the file with the extension of **.sln**. This should cause the project to open and be ready to run or debug as described in the earlier module titled [Getting Started](#).

Run my game program

Click [here](#) to download a zip file containing my version of the program named **XNA0132Proj**. Extract the folder named **XNA0132Proj** from the zip file and save it somewhere on your disk. Start **Visual C# 2010 Express** and select **Open Project...** from the **File** menu. Navigate to the project folder and select the file with the extension of **.sln**. This should cause the project to open and be ready to run or debug as described in the earlier module titled **Getting Started**.

Summary

You learned how to write a simple game program involving user input from the keyboard and the mouse. You also learned how to create on-screen text.

What's next?

At this point, the main tool that we are missing for creating more interesting 2D arcade style games is the ability to incorporate sound effects and music into the games. The research and study of that topic will be left as an exercise for the student.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Xna0132-A Simple Game Program with On-Screen Text
- File: Xna0132.htm
- Published: 02/28/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation :: I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

Note:

Listing 12 . Game1 class for project XNA0132ProjA.

```
/*Project XNA0132ProjA
*****
*****/
using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace XNA0132ProjA {

    public class Game1 :
Microsoft.Xna.Framework.Game {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        SpriteFont Font1;
        Vector2 FontPos;
        //-----
    }
}
```

```

        protected override void LoadContent() {
            spriteBatch = new
SpriteBatch(GraphicsDevice);

            //Create a new SpriteFont object.
            Font1 = Content.Load<SpriteFont>("Lindsey");

            //Create a new Vector2 object to center the
text
            // in the game window.
            FontPos = new Vector2(
                graphics.GraphicsDevice.Viewport.Width
/ 2,
graphics.GraphicsDevice.Viewport.Height / 2);
        } //end LoadContent method
        //-----
        -----//

        protected override void Draw(GameTime
gameTime) {
            GraphicsDevice.Clear(Color.CornflowerBlue);

            spriteBatch.Begin();

            // Draw Lindsey Font
            string output = "Lindsey Font";

            // Find the center of the string
            Vector2 FontOrigin =
Font1.MeasureString(output) / 2;
            //Draw the string with the FontOrigin at the
            // position specified by FontPos.

            spriteBatch.DrawString(
                Font1, //font to use
                output, //output text

```

```

        FontPos,//position re upper left corner
        Color.LightGreen,//color of text
        -45,//rotation of text
        FontOrigin,//place this at FontPos
        2.5f,//scale
        SpriteEffects.None,
        0.5f);//z-order layer

        spriteBatch.End();

        base.Draw(gameTime);
    }//end Draw method
    //-----
-----//

    protected override void Update(GameTime
gameTime) {
        // Allows the game to exit
        if(GamePad.GetState(PlayerIndex.One).
            Buttons.Back ==
ButtonState.Pressed)
            this.Exit();

        //No special update code required.

        base.Update(gameTime);
    }//end Update method
    //-----
-----//

    protected override void Initialize() {
        //Not needed
        base.Initialize();
    }//end Initialize

    //-----
-----//

```

```

    public Game1() { //Typical constructor
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";
    } //end constructor
    //-----
-----//

    protected override void UnloadContent() {
        //Not needed
    } //end UnloadContent
    //-----
-----//
    } //end class
} //end namespace

```

Note:

Listing 13 . Sprite class for project XNA0132Proj.

```

/*Project XNA0132Proj
 * This file defines a Sprite class from which a
Sprite
 * object can be instantiated. This version
supports
 * collision detection based on intersecting
rectangles.
 * It also provides an Edge property that records
and
 * returns the edge number if a sprite collides
with an
 * edge. However, the edge information is
available for
 * only one iteration of the game loop. Normally
the
 * value of Edge is 0. However, it changes to

```

```

1,2,3,or4
    * if a sprite collides with the top, right,
bottom, or
    * left edge of the game window.

*****
*****/

using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;

namespace XNA0132Proj {
    class Sprite {
        private int edge = 0;//new to this version
        private Texture2D image;
        private Vector2 position = new Vector2(0,0);
        private Vector2 direction = new Vector2(0,0);
        private Point windowSize;
        private Random random;
        double elapsedTime;//in milliseconds
        //The following value is the inverse of speed
in
        // moves/msec expressed in msec/move.
        double elapsedTimeTarget;
        //-----
        -----//

        //New to this version.
        //Edge property accessor
        public int Edge {
            get {
                return edge;
            }//end get
        }//end Edge property accessor
    }
}

```

```

//-----
-----//

//Image property accessor
public Texture2D Image {
    get {
        return image;
    }//end get
}//end Image property accessor
//-----
-----//

//Position property accessor
public Vector2 Position {
    get {
        return position;
    }
    set {
        position = value;
    }//end set
}//end Position property accessor
//-----
-----//

//WindowSize property accessor
public Point WindowSize {
    set {
        windowSize = value;
    }//end set
}//end WindowSize property accessor
//-----
-----//

//Direction property accessor
public Vector2 Direction {
    get {
        return direction;
    }

```

```

    }
    set {
        direction = value;
    }//end set
} //end Direction property accessor

//-----
-----//

//Speed property accessor. The set side should
be
// called with speed in moves/msec. The get
side
// returns speed moves/msec.
public double Speed {
    get {
        //Convert from elapsed time in msec/move
to
        // speed in moves/msec.
        return elapsedTimeTarget / 1000;
    }
    set {
        //Convert from speed in moves/msec to
        // elapsed time in msec/move.
        elapsedTimeTarget = 1000 / value;
    } //end set
} //end Speed property accessor
//-----
-----//

//This constructor loads an image for the
sprite
// when it is instantiated. Therefore, it
requires
// an asset name for the image and a reference
to a
// ContentManager object.

```

```

        //Requires a reference to a Random object.
Should
        // use the same Random object for all sprites
to
        // avoid getting the same sequence for
different
        // sprites.
        public Sprite(String assetName,
                        ContentManager contentManager,
                        Random random) {
            image = contentManager.Load<Texture2D>
(assetName);
            image.Name = assetName;
            this.random = random;
        } //end constructor
        //-----
-----//

        //This method can be called to load a new
image
        // for the sprite.
        public void SetImage(String assetName,
                              ContentManager
contentManager) {
            image = contentManager.Load<Texture2D>
(assetName);
            image.Name = assetName;
        } //end SetImage
        //-----
-----//

        //This method causes the sprite to move in the
        // direction of the direction vector if the
elapsed
        // time since the last move exceeds the
elapsed
        // time target based on the specified speed.

```



```

    public void Move(GameTime gameTime) {

        //New to this version
        //Clear the Edge property value. Edge
information
        // is available for only one iteration of
the
        // game loop.
        edge = 0;

        //Accumulate elapsed time since the last
move.
        elapsedTime +=

gameTime.ElapsedGameTime.Milliseconds;

        if(elapsedTime > elapsedTimeTarget) {
            //It's time to make a move. Set the
elapsed
            // time to a value that will attempt to
produce
            // the specified speed on the average.
            elapsedTime -= elapsedTimeTarget;

            //Add the direction vector to the position
            // vector to get a new position vector.
            position =
Vector2.Add(position,direction);

            //Check for a collision with an edge of
the game
            // window. If the sprite reaches an edge,
cause
            // the sprite to wrap around and reappear
at the
            // other edge, moving at the same speed in
a

```

```

        // different direction within the same
quadrant
        // as before. Also set the Edge property
to
        // indicate which edge was involved. 1 is
top, 2
        // is right, 3 is bottom, and 4 is left.
        // Note that the Edge property will be
cleared
        // to 0 the next time the Move method is
called.
        if(position.X < -image.Width) {
            position.X = windowSize.X;
            edge = 4;//collision with the left edge
- new
            NewDirection();
        }//end if

        if(position.X > windowSize.X) {
            position.X = -image.Width / 2;
            edge = 2;//collision with the right edge
- new
            NewDirection();
        }//end if

        if(position.Y < -image.Height) {
            position.Y = windowSize.Y;
            edge = 1;//collision with the top - new
            NewDirection();
        }//end if

        if(position.Y > windowSize.Y) {
            position.Y = -image.Height / 2;
            edge = 3;//collision with the bottom -
new
            NewDirection();
        }//end if on position.Y

```

```

        }//end if on elapsed time
    }//end Move
    //-----
-----//

    //This method determines the length of the
current
    // direction vector along with the signs of
the X
    // and Y components of the current direction
vector.
    // It computes a new direction vector of the
same
    // length with the X and Y components having
random
    // lengths and the same signs.
    //Note that random.NextDouble returns a
    // pseudo-random value, uniformly distributed
    // between 0.0 and 1.0.
    private void NewDirection() {
        //Get information about the current
direction
        // vector.
        double length = Math.Sqrt(
            direction.X *
direction.X +
            direction.Y *
direction.Y);
        Boolean xNegative =
            (direction.X < 0) ? true :
false;
        Boolean yNegative =
            (direction.Y < 0) ? true :
false;

        //Compute a new X component as a random
portion of

```

```

        // the vector length.
        direction.X =
            (float)(length *
random.NextDouble());
        //Compute a corresponding Y component that
will
        // keep the same vector length.
        direction.Y = (float)Math.Sqrt(length *
length -
            direction.X *
direction.X);

        //Set the signs on the X and Y components to
match
        // the signs from the original direction
vector.
        if(xNegative)
            direction.X = -direction.X;
        if(yNegative)
            direction.Y = -direction.Y;
    }//end NewDirection
    //-----
    -----//

    public void Draw(SpriteBatch spriteBatch) {
        //Call the simplest available version of
        // spriteBatch.Draw

spriteBatch.Draw(image, position, Color.White);
    }//end Draw method
    //-----
    -----//

    //Returns the current rectangle occupied by
the
    // sprite.
    public Rectangle GetRectangle() {

```

[illegible]

```

image.Height);
    Rectangle targetRectangle;
    int cnt = 0;

    while(cnt < target.Count) {
        targetRectangle =
target[cnt].GetRectangle();

if(thisRectangle.Intersects(targetRectangle)) {
    return target[cnt];
} //end if
    cnt++;
} //end while loop

    return null; //no collision detected
} //end IsCollision
//-----
-----//

    } //end class
} //end namespace

```

Note:

Listing 14 . Game1 class for project XNA0132Proj.

```

/*Project XNA0132Proj
 * This project demonstrates how to write a simple
2D
 * arcade style game. Ten spiders try to make it
across
 * a web from top to bottom in opposition to two
 * ladybugs. If a ladybug collides with a spider,
the

```

```
* spider is eaten by the ladybug.  
*  
* The ladybugs can be moved either with the  
keyboard or  
* the mouse. Press the arrow keys to move one of  
the  
* ladybugs. Press the A key plus the arrow keys  
to move  
* the other ladybug.  
*  
* Press the left mouse button to drag one of the  
* ladybugs with the mouse. Press the right arrow  
key  
* to drag the other ladybug with the mouse.  
*  
* If a spider makes it from the top to the bottom  
of  
* the game window, it wraps back to the top and  
starts  
* the trip over.  
*  
* On-screen text keeps track of the number of  
spider  
* crossings at the bottom of the game window.  
Note,  
* however, that if two spiders cross the bottom  
in  
* very close proximity, they may not both get  
counted.  
*  
* This program demonstrates how to display on-  
screen  
* text. Note however that it is necessary to  
create  
* a font resource before you can display onscreen  
text.  
*
```

```

*****
***/
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using XNA0132Proj;

namespace XNA0132Proj {

    public class Game1 :
Microsoft.Xna.Framework.Game {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        //Use the following values to set the size of
the
        // client area of the game window. The actual
window
        // with its frame is somewhat larger depending
on
        // the OS display options. On my machine with
its
        // current display options, these dimensions
        // produce a 1024x768 game window.
        int windowWidth = 1017;
        int windowHeight = 738;

        //This is the length of the greatest distance
in
        // pixels that any sprite will move in a
single
        // frame of the game loop.
        double maxVectorLength = 5.0;

        Sprite spiderWeb;//reference to a background

```


sprite.

```
//References to the spiders are stored in this
// List object.
List<Sprite> spiders = new List<Sprite>();
int numSpiders = 10;//Number of spiders.
//The following value should never exceed 60
moves
// per second unless the default frame rate is
also
// increased to more than 60 frames per
second.
double maxSpiderSpeed = 30;//moves per second

//References to the Ladybugs are stored in
this List.
List<Sprite> ladybugs = new List<Sprite>();
int numLadybugs = 2;//Max number of ladybugs

//Random number generator. It is best to use a
single
// object of the Random class to avoid the
// possibility of using different streams that
// produce the same sequence of values.
//Note that the random.NextDouble() method
produces
// a pseudo-random value where the sequence of
values
// is uniformly distributed between 0.0 and
1.0.
Random random = new Random();

//The following variable is used to count the
number
// of spiders that make it past the ladybugs
and
```

```

// reach the bottom of the game window.
int spiderCount = 0;

SpriteFont Font1;//font for on-screen text
Vector2 FontPos;//position of on-screen text
//-----
-----//

public Game1() { //constructor
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";

    //Set the size of the game window.
    graphics.PreferredBackBufferWidth =
windowWidth;
    graphics.PreferredBackBufferHeight =
windowHeight;
} //end constructor
//-----
-----//

protected override void Initialize() {
    //No initialization required.
    base.Initialize();
} //end Initialize
//-----
-----//

protected override void LoadContent() {
    spriteBatch = new
SpriteBatch(GraphicsDevice);
    Font1 = Content.Load<SpriteFont>
("Kootenay");
    IsMouseVisible = true; //make mouse visible

    //Create a sprite for the background image.

```

```

        spiderWeb =
            new
Sprite("spiderwebB",Content,random);
        spiderWeb.Position = new Vector2(0f,0f);

        //Instantiate all of the spiders and cause
them to
        // move from left to right, top to
        // bottom. Pass a reference to the same
Random
        // object to all of the sprites.
        for(int cnt = 0;cnt < numSpiders;cnt++) {
            spiders.Add(
                new
Sprite("blackWidowSpider",Content,random));

                //Set the position of the current spider
at a
                // random location within the game window
but
                // near the top of the game window.
                spiders[cnt].Position = new Vector2(
                    (float)(windowWidth *
random.NextDouble()),
                    (float)((windowHeight/10) *
random.NextDouble()));

                //Get a direction vector for the current
spider.
                // Make both components positive to cause
the
                // vector to point down and to the right.
                spiders[cnt].Direction = DirectionVector(
                    (float)maxVectorLength,
                    (float)(maxVectorLength *

```

```

random.NextDouble()),
        false, //xNeg
        false); //yNeg

        //Notify the spider object of the size of
the
        // game window.
        spiders[cnt].WindowSize =
            new
Point(windowWidth, windowHeight);

        //Set the speed in moves per second for
the
        // current spider to a random value
between
        // maxSpiderSpeed/2 and maxSpiderSpeed.
        spiders[cnt].Speed = maxSpiderSpeed / 2
            + maxSpiderSpeed *
random.NextDouble() / 2;
    } //end for loop

    //Instantiate all of the ladybugs. They move
under
    // control of the keyboard or the mouse.
    for(int cnt = 0; cnt < numLadybugs; cnt++) {
        ladybugs.Add(
            new
Sprite("ladybug", Content, random));

        //Position the ladybugs at a random
position
        // near the bottom of the game window.
        ladybugs[cnt].Position = new Vector2(
            (float)(windowWidth *
random.NextDouble()),
            (float)(windowHeight -

```

```

ladybugs[cnt].Image.Height));
    }//end for loop

    //Position the on-screen text.
    FontPos = new Vector2(windowWidth / 2, 50);

    //Position the mouse pointer in the center
of the
    // game window.
    Mouse.SetPosition(windowWidth /
2>windowHeight /2);

    }//end LoadContent
    //-----
-----//

    //This method returns a direction vector given
the
    // length of the vector, the length of the
    // X component, the sign of the X component,
and the
    // sign of the Y component. Set negX and/or
negY to
    // true to cause them to be negative. By
adjusting
    // the signs on the X and Y components, the
vector
    // can be caused to point into any of the four
    // quadrants.
    private Vector2 DirectionVector(float vecLen,
                                    float xLen,
                                    Boolean negX,
                                    Boolean negY)
{
    Vector2 result = new Vector2(xLen,0);
    result.Y = (float)Math.Sqrt(vecLen * vecLen
                                - xLen *

```

```

xLen);
    if(negX)
        result.X = -result.X;
    if(negY)
        result.Y = -result.Y;
    return result;
} //end DirectionVector
//-----
-----//

protected override void UnloadContent() {
    //No content unload required.
} //end unloadContent
//-----
-----//

protected override void Update(GameTime
gameTime) {
    //Tell all the spiders in the list to move.
    for(int cnt = 0; cnt < spiders.Count; cnt++) {
        spiders[cnt].Move(gameTime);
    } //end for loop

    //Tell each ladybug to test for a collision
with a
    // spider and to return a reference to the
spider
    // if there is a collision. Return null if
there is
    // no collision.
    for(int cnt = 0; cnt < ladybugs.Count; cnt++)
{
    //Test for a collision between this
ladybug and
    // all of the spiders in the list of
spiders.

```

```

        Sprite target =
ladybugs[cnt].IsCollision(spiders);
        if(target != null) {
            //There was a collision. The spider gets
eaten.
            // Remove it from the list of spiders.
            spiders.Remove(target);
        }//end if
    }//end for loop

    //Check to see if any spiders have made it
to the
    // bottom edge.
    for(int cnt = 0;cnt < spiders.Count;cnt++) {
        if(spiders[cnt].Edge == 3)
            //One or more made it to the bottom
edge.
            spiderCount += 1;
    }//end for loop

    //The following code is used to move one or
the
    // other of two ladybugs using the arrow
keys.
    // Press only the arrow keys to move one of
the
    // ladybugs. Press the A or a plus the arrow
keys
    // to move the other ladybug.
    //The ladybugs cannot be moved outside the
game
    // window.
    //When an arrow key is pressed, the ladybug
moves
    //five pixels per call to the Update method.

```

```

        //Get the state of the keyboard.
        KeyboardState keyboardState =
Keyboard.GetState();

        //Execute moves on one ladybug with arrow
keys plus
        // the A or a key.
        if(keyboardState.IsKeyDown(Keys.Left) &&
            (keyboardState.IsKeyDown(Keys.A)) &&
            (ladybugs[0].Position.X > 0)) {
            ladybugs[0].Position = new Vector2(
ladybugs[0].Position.X - 5,
ladybugs[0].Position.Y);
        }//end if

        if(keyboardState.IsKeyDown(Keys.Right) &&
            (keyboardState.IsKeyDown(Keys.A)) &&
            (ladybugs[0].Position.X <
            (windowWidth -
ladybugs[1].Image.Width))) {
            ladybugs[0].Position = new Vector2(
ladybugs[0].Position.X + 5,
ladybugs[0].Position.Y);
        }//end if

        if(keyboardState.IsKeyDown(Keys.Up) &&
            (keyboardState.IsKeyDown(Keys.A)) &&
            (ladybugs[0].Position.Y > 0)) {
            ladybugs[0].Position = new Vector2(
ladybugs[0].Position.X,
ladybugs[0].Position.Y - 5);

```



```

    }//end if

    if(keyboardState.IsKeyDown(Keys.Down) &&
        (keyboardState.IsKeyDown(Keys.A)) &&
        (ladybugs[0].Position.Y <
            (windowHeight -
ladybugs[1].Image.Height))) {
        ladybugs[0].Position = new Vector2(
ladybugs[0].Position.X,
ladybugs[0].Position.Y + 5);
    }//end if

    //Execute moves on the other ladybug with
arrow
    // keys pressed but the A key not pressed.
    if(keyboardState.IsKeyDown(Keys.Left) &&
        !(keyboardState.IsKeyDown(Keys.A)) &&
        (ladybugs[1].Position.X > 0)) {
        ladybugs[1].Position = new Vector2(
ladybugs[1].Position.X - 5,
ladybugs[1].Position.Y);
    }//end if

    if(keyboardState.IsKeyDown(Keys.Right) &&
        !(keyboardState.IsKeyDown(Keys.A)) &&
        (ladybugs[1].Position.X <
            (windowWidth -
ladybugs[1].Image.Width))) {
        ladybugs[1].Position = new Vector2(
ladybugs[1].Position.X + 5,
ladybugs[1].Position.Y);

```

```

        }//end if

        if(keyboardState.IsKeyDown(Keys.Up) &&
            !(keyboardState.IsKeyDown(Keys.A)) &&
            (ladybugs[1].Position.Y > 0)) {
            ladybugs[1].Position = new Vector2(
ladybugs[1].Position.X,
ladybugs[1].Position.Y - 5);
        }//end if

        if(keyboardState.IsKeyDown(Keys.Down) &&
            !(keyboardState.IsKeyDown(Keys.A)) &&
            (ladybugs[1].Position.Y <
            (windowHeight -
ladybugs[1].Image.Height))) {
            ladybugs[1].Position = new Vector2(
ladybugs[1].Position.X,
ladybugs[1].Position.Y + 5);
        }//end if

        //The following code is used to drag one or
the
        // other of two ladybugs using the mouse.
Press
        // the left mouse button to drag one of the
        // ladybugs. Press the right mouse button to
drag
        // the other ladybug.

        //Get the state of the mouse.
        MouseState mouseState = Mouse.GetState();

```

```

        //Press the left mouse button to move one
ladybug.
        if(mouseState.LeftButton ==
ButtonState.Pressed) {
            ladybugs[0].Position =
                new
Vector2(mouseState.X,mouseState.Y);
        }//end if

        //Press the right mouse button to move the
other
        // ladybug.
        if(mouseState.RightButton ==
ButtonState.Pressed) {
            ladybugs[1].Position =
                new
Vector2(mouseState.X,mouseState.Y);
        }//end if


        base.Update(gameTime);
    }//end Update method
    //-----
-----//

    protected override void Draw(GameTime
gameTime) {

        spriteBatch.Begin();

        spiderWeb.Draw(spriteBatch);//draw
background

        //Draw all spiders.
        for(int cnt = 0;cnt < spiders.Count;cnt++) {
            spiders[cnt].Draw(spriteBatch);
        }//end for loop

```

```

        //Draw all ladybugs.
        for(int cnt = 0;cnt < ladybugs.Count;cnt++)
    {
        ladybugs[cnt].Draw(spriteBatch);
    }//end for loop

    //Draw the output text.
    string output = "";
    if(spiderCount == 0){
        output = "Congratulations. No spiders made
it to"
            + " the bottom.";
    }else{
        output = "Oops, " + spiderCount + " or
more "
            + "spiders made it to the
bottom.";
    }//end else

    // Find the center of the string
    Vector2 FontOrigin =
Font1.MeasureString(output) / 2;
    // Draw the string
    spriteBatch.DrawString(Font1,
                            output,
                            FontPos,
                            Color.Yellow,
                            0,//angle
                            FontOrigin,
                            1.0f,//scale
                            SpriteEffects.None,
                            0.0f);//layer depth

    spriteBatch.End();

```

```
        base.Draw(gameTime);  
    }//end Draw method  
    //-----  
-----//  
    }//end class  
}//end namespace
```

-end-